USENIX Association

Proceedings of the
23rd USENIX Security Symposium

August 20–22, 2014
San Diego, CA

Conference Organizers

Program Chair
Kevin Fu, *University of Michigan*

Deputy Program Chair
Jaeyeon Jung, *Microsoft Research*

Program Committee
Bill Aiello, *University of British Columbia*
Steven Bellovin, *Columbia University*
Emery Berger, *University of Massachusetts Amherst*
Dan Boneh, *Stanford University*
Nikita Borisov, *University of Illinois at Urbana-Champaign*
David Brumley, *Carnegie Mellon University*
Kevin Butler, *University of Oregon*
Srdjan Capkun, *ETH Zürich*
Stephen Checkoway, *Johns Hopkins University*
Nicolas Christin, *Carnegie Mellon University*
George Danezis, *University College London*
Srini Devadas, *Massachusetts Institute of Technology*
Roger Dingledine, *The Tor Project*
David Evans, *University of Virginia*
Nick Feamster, *Georgia Institute of Technology*
Adrienne Porter Felt, *Google*
Simson Garfinkel, *Naval Postgraduate School*
Virgil Gligor, *Carnegie Mellon University*
Rachel Greenstadt, *Drexel University*
Steve Gribble, *University of Washington and Google*
Carl Gunter, *University of Illinois at Urbana-Champaign*
Alex Halderman, *University of Michigan*
Nadia Heninger, *University of Pennsylvania*
Thorsten Holz, *Ruhr-University Bochum*
Jean-Pierre Hubaux, *École Polytechnique Fédérale de Lausanne*
Cynthia Irvine, *Naval Postgraduate School*
Jaeyeon Jung, *Microsoft Research*
Chris Kanich, *University of Illinois at Chicago*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *Microsoft Research and University of Washington*
Farinaz Koushanfar, *Rice University*
Zhenkai Liang, *National University of Singapore*
David Lie, *University of Toronto*
Stephen McCamant, *University of Minnesota*
Damon McCoy, *George Mason University*
Patrick McDaniel, *Pennsylvania State University*
Cristina Nita-Rotaru, *Purdue University*
Zachary N. J. Peterson, *California Polytechnic State University*
Raj Rajagopalan, *Honeywell Labs*
Ben Ransford, *University of Washington*
Thomas Ristenpart, *University of Wisconsin—Madison*
Prateek Saxena, *National University of Singapore*
Patrick Schaumont, *Virginia Polytechnic Institute and State University*
Stuart Schechter, *Microsoft Research*
Simha Sethumadhavan, *Columbia University*
Cynthia Sturton, *University of North Carolina at Chapel Hill*
Wade Trappe, *Rutgers University*
Eugene Y. Vasserman, *Kansas State University*
Ingrid Verbauwhede, *Katholieke Universiteit Leuven*
Giovanni Vigna, *University of California, Santa Barbara*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*
Rui Wang, *Microsoft Research*
Matthew Wright, *University of Texas at Arlington*
Wenyuan Xu, *University of South Carolina*

Invited Talks Committee
Sandy Clark, University of Pennsylvania
Matthew Green, Johns Hopkins University
Thorsten Holz, *Ruhr-University Bochum*
Ben Laurie, *Google*
Damon McCoy, *George Mason University*
Jon Oberheide, *Duo Security*
Patrick Traynor (Chair), *University of Florida*

Poster Session Coordinator
Franziska Roesner, *University of Washington*

Steering Committee
Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Casey Henderson, *USENIX*
Tadayoshi Kohno, *University of Washington*
Fabian Monrose, *University of North Carolina, Chapel Hill*
Niels Provos, *Google*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*

External Reviewers

Fardin Abdi Taghi Abad
Shabnam Aboughadareh
Sadia Afroz
Devdatta Akhawe
Mahdi N. Al-Ameen
Thanassis Avgerinos
Erman Ayday
Guangdong Bai
Josep Balasch
Adam Bates
Felipe Beato
Robert Beverly
Antonio Bianchi
Igor Bilogrevic
Vincent Bindschaedler
Eric Bodden
Jonathan Burket
Aylin Caliskan-Islam
Henry Carter
Sze Yiu Chan
Peter Chapman
Longze Chen
Shuo Chen
Yangyi Chen
Yingying Chen
Sonia Chiasson
Zheng Leong Chua
Sandy Clark
Shane Clark
Charlie Curtsinger
Drew Davidson
Soteris Demetriou
Lucas Devi
Anh Ding
Xinshu Dong
Zheng Dong
Manuel Egele
Ari Feldman
Bryan Ford
Matt Fredrikson
Afshar Ganjali
Christina Garman
Behrad Garmany
Robert Gawlik
Benedikt Gierlichs
Matt Green
Christian Grothoff
Weili Han
S M Taiabul Haque
Cormac Herley
Stephan Heuser
Matthew Hicks
Daniel Holcomb
Brandon Holt
Peter Honeyman
Endadul Hoque
Amir Houmansadr
Hong Hu
Yan Huang
Kevin Huguenin
Mathias Humbert
Thomas Hupperich
Nathaniel Husted
Yoshi Imamoto
Luca Invernizzi
Sam Jero
Limin Jia
Yaoqi Jia
Zhaopeng Jia
Aaron Johnson
Marc Juarez
Min Suk Kang
Georg Koppen
Karl Koscher
Marc Kührer
Hyojeong Lee
Nektarios Leontiadis
Wenchao Li
Xiaolei Li
Zhou Li
Hoon Wei Lim
Haiyang Liu
Lilei Lu
Loi Luu
Mingjie Ma
Alex Malozemoff
Jian Mao
Claudio Marforio
Paul Martin
Markus Miettinen
Andrew Miller
Apurva Mohan
Andres Molina-Markham
Benjamin Mood
Tyler Moore
Chris Morrow
Thomas Moyer
Jan Tobias Muehlberg
Collin Mulliner
Arslan Munir
Muhammad Naveed
Damien Octeau
Temitope Oluwafemi
Xinming Ou
Rebekah Overdorf
Xiaorui Pan
Roel Peeters
Rahul Potharaju
Niels Provos
Daiyong Quan
Moheeb Abu Rajab
Siegfried Rasthofer
Brad Reaves
Jennifer Rexford
Alfredo Rial
Christian Rossow
Mastooreh Salajegheh
Nitesh Saxena
Dries Schellekens
Edward J. Schwartz
Stefaan Seys
Ravinder Shankesi
Shweta Shinde
Dave Singelee
Ian Smith
Kyle Soska
Raoul Strackx
Gianluca Stringhini
Shruti Tople
Emma Tosch
Patrick Traynor
Sebastian Uellenbeck
Frederik Vercauteren
Timothy Vidas
John Vilk
Paul Vines
Chao Wang
Zachary Weinberg
Steve Weis
Marcel Winandy
Michelle Wong
Maverick Woo
Eric Wustrow
Luojie Xiang
Luyi Xing
Hui Xue
Miao Yu
Kan Yuan
Samee Zahur
Jun Zhao
Xiaoyong Zhou
Yuchen Zhou
Zongwei Zhou

23rd USENIX Security Symposium
August 20–22, 2014
San Diego, CA

Wednesday, August 20, 2014

Privacy

Mass Pwnage

Privacy Enhancing Technology

Crime and Pun.../Measure-ment

Thursday, August 21, 2014

Forensics

Attacks and Transparency

(Thursday, August 21, continues on next page)

ROP: Return of the %edi

Safer Sign-Ons

Tracking Targeted Attacks against Civilians and NGOs

Passwords

Web Security: The Browser Strikes Back

Friday, August 22, 2014

Side Channels

After Coffee Break Crypto

Program Analysis: Attack of the Codes

(Friday, August 22, continues on next page)

After Lunch Break Crypto

Program Analysis: A New Hope

Mobile Apps and Smart Phones

Message from the 23rd USENIX Security Symposium Program Chair

Welcome to the 23rd USENIX Security Symposium in San Diego, CA! If I were to pick one phrase to describe this year, it would be "record-breaking" with "server-breaking" as a close second. This year's program will literally knock your socks off, especially the session on transparency. The Program Committee received 350 submissions, a whopping 26% increase over last year. After careful deliberation with an emphasis on positive and constructive reviewing, we accepted 67 papers. I'd like to thank the authors, the invited speakers, the Program Committee members and other organizers, the external reviewers, the sponsors, the USENIX staff, and the attendees for continuing to make USENIX Security a premier venue for security and privacy research. I'd also like to welcome our newcomers—be they students or seasoned researchers—to make the most of our technical sessions and evening events.

The USENIX Security Program Committee has followed a procedure that has changed little since the late 1990s, even as the number of submissions has grown from several dozen to hundreds. I evaluated the attendee survey of the previous year, interviewed both elders and newcomers, and performed a linear regression on paper submission statistics collected over the last 15 years. As a result, I instituted a number of changes to cope with the growth and maturation of our field. The changes include: introducing the light/heavy PC model to USENIX Security, the shadow PC concept, a hybrid unblinding process during the final stages of reviewing, a Doctoral Colloquium for career building, a Lightning Talks session of short videos, and enforcement of positive and constructive reviewing methodologies during deliberation.

I initially invited 56 Program Committee members (in honor of DES) to cover the advertised topics while attempting to increase diversity across dimensions of country, gender, geography, institution, and seniority. Each paper reaching the discussion phase had multiple reviewers who could speak at the PC meeting having read the paper. The authors of submissions were not revealed to the reviewers until late in the decision-making phase, and every paper was reviewed by at least two reviewers. The median number of reviews per paper was five. The heavy Program Committee met to discuss the submissions in May at the University of Michigan in Ann Arbor, Michigan. Attendees enjoyed quintessential and memorable Ann Arbor cuisine, including NSA-themed cocktails provided by Duo Security. Ahead of the meeting, each paper was assigned a discussion lead. We used punch cards and the "meeting tracker" feature of the HotCRP software so that each paper received ten minutes of discussion in an order planned ahead of time to ensure efficient and fairly allocated deliberation time. We finished exactly on schedule after two days.

The entire Program Committee invested a tremendous effort in reviewing and discussing these papers. PC members submitted on average 23 reviews. They received no tangible rewards other than a complimentary webcam privacy lens cap. So please, thank the all-volunteer PC members and external reviewers for their countless hours of work.

I would like to thank Patrick Traynor for chairing the Invited Talks Committee that selected a thought-provoking set of invited talks. The Poster and Works-in-Progress sessions are also "can't miss" events for glimpses of cutting-edge and emerging research activities. I would like to thank Franziska Roesner for serving as the Poster Session Chair, and Tadayoshi Kohno for serving as both the WiPs Chair and Shadow PC Coordinator.

I am especially appreciative of Jaeyeon Jung for accepting the responsibilities of Deputy PC Chair after I had to deal with an unexpected disaster and relocation of my children after our house collapsed. When evaluating excuses for late reviews, I would consider whether the PC member's house had recently collapsed. I also thank the USENIX Security Steering Committee and Niels Provos in particular for serving as USENIX Liaison. Eddie Kohler earns MVP status for his fast responses to various HotCRP questions. Finally, I would like to thank all of the authors who submitted their research for consideration. Our community is in its prime, so please enjoy the 23rd USENIX Security Symposium!

Kevin Fu, ***University of Michigan***
USENIX Security '14 Program Chair

Privee: An Architecture for Automatically Analyzing Web Privacy Policies

Sebastian Zimmeck and Steven M. Bellovin

Department of Computer Science, Columbia University
{*sebastian,smb*}*@cs.columbia.edu*

Abstract

Privacy policies on websites are based on the notice-and-choice principle. They notify Web users of their privacy choices. However, many users do not read privacy policies or have difficulties understanding them. In order to increase privacy transparency we propose Privee—a software architecture for analyzing essential policy terms based on crowdsourcing and automatic classification techniques. We implement Privee in a proof of concept browser extension that retrieves policy analysis results from an online privacy policy repository or, if no such results are available, performs automatic classifications. While our classifiers achieve an overall F-1 score of 90%, our experimental results suggest that classifier performance is inherently limited as it correlates to the same variable to which human interpretations correlate—the ambiguity of natural language. This finding might be interpreted to call the notice-and-choice principle into question altogether. However, as our results further suggest that policy ambiguity decreases over time, we believe that the principle is workable. Consequently, we see Privee as a promising avenue for facilitating the notice-and-choice principle by accurately notifying Web users of privacy practices and increasing privacy transparency on the Web.

1 Introduction

Information privacy law in the U.S. and many other countries is based on the free market notice-and-choice principle [28]. Instead of statutory laws and regulations, the privacy regime is of a contractual nature—the provider of a Web service posts a privacy policy, which a user accepts by using the site. In this sense, privacy policies are fundamental building blocks of Web privacy. The Federal Trade Commission (FTC) strictly enforces companies' violations of their promises in privacy policies. However, only few users read privacy policies and those who do find them oftentimes hard to understand [58]. The resulting information asymmetry leaves users uninformed about their privacy choices [58], can lead to market failure [57], and ultimately casts doubt on the notice-and-choice principle.

Various solutions were proposed to address the problem. However, none of them gained widespread acceptance—neither in the industry, nor among users. Most prominently, The Platform for Privacy Preferences (P3P) project [29, 32] was not widely adopted, mainly, because of a lack of incentive on part of the industry to express their policies in P3P format. In addition, P3P was also criticized for not having enough expressive power to describe privacy practices accurately and completely [28, 11]. Further, existing crowdsourcing solutions, such as Terms of Service; Didn't Read (ToS;DR) [5], may not scale well and still need to gain more popularity. Informed by these experiences, which we address in more detail in Section 2, we present Privee—a novel software architecture for analyzing Web privacy policies. In particular, our contributions are:

- the Privee concept that combines rule and machine learning (ML) classification with privacy policy crowdsourcing for seamless integration into the existing privacy regime on the Web (Section 3);
- an implementation of Privee in a Google Chrome browser extension that interacts with privacy policy websites and the ToS;DR repository of crowdsourced privacy policy results (Section 4);
- a statistical analysis of our experimental results showing that the ambiguity of privacy policies makes them inherently difficult to understand for both humans and automatic classifiers (Section 5);
- pointers for further research on notice-and-choice and adaptations that extend Privee as the landscape of privacy policy analysis changes and develops (Section 6).

2 Related Work

While only few previous works are directly applicable, our study is informed by four areas of previous research: privacy policy languages (Section 2.1), legal information extraction (Section 2.2), privacy policy crowdsourcing (Section 2.3), and usable privacy (Section 2.4).

2.1 Privacy Policy Languages

Initial work on automatic privacy policy analysis focused on making privacy policies machine-readable. That way a browser or other user agent could read the policies and alert the user of good and bad privacy practices. Reidenberg [67] suggested early on that Web services should represent their policies in the Platform for Internet Content Selection (PICS) format [10]. This and similar suggestions lead to the development of P3P [29, 32], which provided a machine-readable language for specifying privacy policies and displaying their content to users [33]. To that end, the designers of P3P implemented various end users tools, such as Privacy Bird [30], a browser extension for Microsoft's Internet Explorer that notifies users of the privacy practices of a Web service whose site they visit, and Privacy Bird Search [24], a P3P-enabled search engine that returns privacy policy information alongside search results.

The development of P3P was complemented by various other languages and tools. Of particular relevance was A P3P Preference Privacy Exchange Language (APPEL) [31], which enabled users to express their privacy preferences vis-à-vis Web services. APPEL was further extended in the XPath project [14] and inspired the User Privacy Policy (UPP) language [15] for use in social networks. For industry use, the Platform for Enterprise Privacy Practices (E-P3P) [47] was developed allowing service providers to formulate, supervise, and enforce privacy policies. Similar languages and frameworks are the Enterprise Privacy Authorization Language (EPAL) [18], the SPARCLE Policy Workbench [22, 23], Jeeves [78], and XACML [12]. However, despite all efforts the adoption rate of P3P policies among Web services remained low [11], and the P3P working group was closed in 2006 due to lack of industry participation [28].

Instead of creating new machine-readable privacy policy formats we believe that it is more effective to use what is already there—privacy policies in natural language. The reasons are threefold: First, natural language is the de-facto standard for privacy policies on the Web, and the P3P experience shows that there is currently no industry-incentive to move to a different standard. Second, U.S. governmental agencies are in strong support of the natural language format. In particular, the FTC, the main privacy regulator, called for more industry-efforts to increase policy standardization and comprehensibility [38]. Another agency, the National Science Foundation, awarded $3.75 million to the Usable Privacy Policy Project [9] to explore possibilities of automatic policy analysis. Third, natural language has stronger expressive power compared to a privacy policy language. It allows for industry-specific formulation of privacy practices and accounts for the changing legal landscape over time.

2.2 Legal Information Extraction

Given our decision to make use of natural language policies, the question becomes how salient information can be extracted from unordered policy texts. While most works in legal information extraction relate to domains other than privacy, they still provide some guidance. For example, Westerhout et al. [75, 76] had success in combining a rule-based classifier with an ML classifier to identify legal definitions. In another line of work de Maat et al. [35, 36] aimed at distinguishing statutory provisions according to types (such as procedural rules or appendices) and patterns (such as definitions, rights, or penal provisions). They concluded that it was unnecessary to employ something more complex than a simple pattern recognizer [35, 36]. Other tasks focused on the extraction of information from statutory and regulatory laws [21, 20], the detection of legal arguments [59], or the identification of case law sections [54, 71].

To our knowledge, the only works in the privacy policy domain are those by Ammar et al. [16], Costante et al. [26, 27], and Stamey and Rossi [70]. As part of the Usable Privacy Policy Project [9] Ammar et al. presented a pilot study [16] with a narrow focus on classifying provisions for the disclosure of information to law enforcement officials and users' rights to terminate their accounts. They concluded the feasibility of natural language analysis in the privacy policy domain in general. In their first work [26] Costante et al. used general natural language processing libraries to evaluate the suitability of rule-based identification of different types of user information that Web services collect. Their results are promising and indicate the feasibility of rule-based classifiers. In a second work [27] Costante et al. selected an ML approach for assessing whether privacy policies cover certain subject matters. Finally, Stamey and Rossi [70] provided a program for identifying ambiguous words in privacy policies.

The discussed works [16, 26, 27, 70] confirm the suitability of rule and ML classifiers in the privacy policy domain. However, neither provides a comprehensive concept, nor addresses, for example, how to process the policies or how to make use of crowdsourcing results. The latter point is especially important because, as shown in Section 5, automatic policy classification on its own is in-

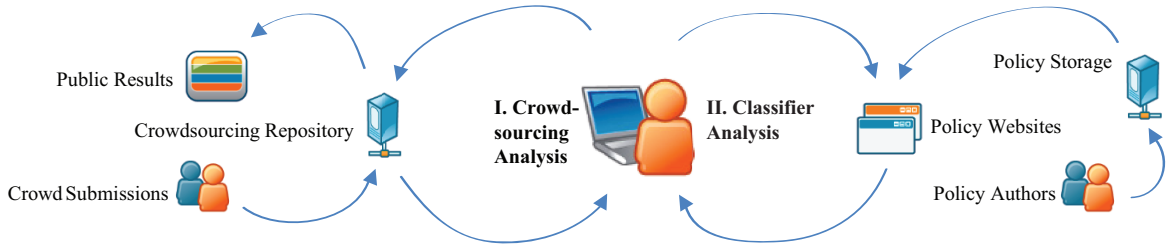


Figure 1: Privee overview. When a user requests a privacy policy analysis, the program checks whether the analysis results are available at a crowdsourcing repository (to which crowd contributors can submit analysis results of policies). If results are available, they are returned and displayed to the user (I. Crowdsourcing Analysis). If no results are available, the policy text is fetched from the policy website, analyzed by automatic classifiers on the client machine, and then the analysis results are displayed to the user (II. Classifier Analysis).

herently limited. In addition, as the previous works' purpose is to generally show the viability of natural language privacy policy analysis, they are constrained to classifying one or two individual policy terms or features. As they process each classification task separately, there was also no need to address questions of handling multiple classifiers or discriminating which extracted features belong to which classification task. Because of their limited scope none of the previous works relieves the user from actually reading the analyzed policy. In contrast, it is our goal to provide users with a privacy policy summary in lieu of the full policy. We want to condense a policy into essential terms, make it more comprehensible, provide guidance on the analyzed practices, and give an overall evaluation of its privacy level.

2.3 Privacy Policy Crowdsourcing

There are various crowdsourcing repositories where crowd contributors evaluate the content of privacy policies and submit their results into a centralized collection for publication on the Web. Sometimes policies are also graded. Among those repositories are ToS;DR [5], privacychoice [4], TOSBack [7], and TOSBack2 [8]. Crowdsourcing has the advantage that it combines the knowledge of a large number of contributors, which, in principle, can lead to much more nuanced interpretations of ambiguous policy provisions than current automatic classifiers could provide. However, all crowdsourcing approaches suffer from a lack of participation and, consequently, do not scale well. While the analysis results of the most popular websites may be available, those for many lesser known sites are not. In addition, some repositories only provide the possibility to look up the results on the Web without offering convenient user access, for example, by means of a browser extension or other software.

2.4 Usable Privacy

Whether the analysis of a privacy policy is based on crowdsourcing or automatic classifications, in order to notify users of the applicable privacy practices it is not enough to analyze policy content, but rather the results must also be presented in a comprehensible, preferably, standardized format [60]. In this sense, usable privacy is orthogonal to the other related areas: no matter how the policies are analyzed, a concise, user-friendly notification is always desirable. In particular, privacy labels may help to succinctly display privacy practices [48, 49, 51, 65, 66]. Also, privacy icons, such as those proposed by PrimeLife [39, 45], KnowPrivacy [11], and the Privacy Icons project [3], can provide visual clues to users. However, care must be taken that the meaning of the icons is clear to the users [45]. In any case, it should be noted that while usability is an important element of the Privee concept, we have not done a usability study for our Privee extension as it is just a proof of concept.

3 The Privee Concept

Figure 1 shows a conceptual overview of Privee. Privee makes use of automatic classifiers and complements them with privacy policy crowdsourcing. It integrates all components of the current Web privacy ecosystem. Policy authors write their policies in natural language and do not need to adopt any special machine-readable policy format. While authors certainly can express the same semantics as with P3P, which we demonstrate in Section 4.6.2, they can also go beyond and use their language much more freely and naturally.

When a user wants to analyze a privacy policy, Privee leverages the discriminative power of crowdsourcing. As we will see in Section 5 that classifiers and human interpretations are inherently limited by ambiguous language,

it is especially important to resolve those ambiguities by providing a forum for discussion and developing consensus among many crowd contributors. Further, Privee complements the crowdsourcing analysis with the ubiquitous applicability of rule and ML classifiers for policies that are not yet analyzed by the crowd. Because the computational requirements are low, as shown in Section 5.3, a real time analysis is possible.

As the P3P experience showed [28] that a large fraction of Web services with P3P policies misrepresented their privacy practices, presumably in order to prevent user agents from blocking their cookies, any privacy policy analysis software must be guarded against manipulation. However, natural language approaches, such as Privee, have an advantage over P3P and other machine-readable languages. Because it is not clear whether P3P policies are legally binding [69] and the FTC never took action to enforce them [55], the misrepresentation of privacy practices in those policies is a minor risk that many Web services are willing to take. This is true for other machine-readable policy solutions as well. In contrast, natural language policies can be valid contracts [1] and are subject to the FTC's enforcement actions against unfair or deceptive acts or practices (15 U.S.C. §45(a)(1)). Thus, we believe that Web services are more likely to ensure that their natural language policies represent their practices accurately.

Given that natural language policies attempt to truly reflect privacy practices, it is important that the policy text is captured completely and without additional text, in particular, free from advertisements on the policy website. Further, while it is true that an ill-intentioned privacy policy author might try to deliberately use ambiguous language to trick the classifier analysis, this strategy can only go so far as ambiguous contract terms are interpreted against the author (Restatement (Second) of Contracts, §206) and might also cause the FTC to challenge them as unfair or deceptive. Beyond safeguarding the classifier analysis, it is also important to prevent the manipulation of the crowdsourcing analysis. In this regard, the literature on identifying fake reviews should be brought to bear. For example, Wu et al. [77] showed that fake reviews can be identified by a suspicious grade distribution and their posting time following negative reviews. In order to ensure that the crowdsourcing analysis returns the latest results the crowdsourcing repository should also keep track of privacy policy updates.

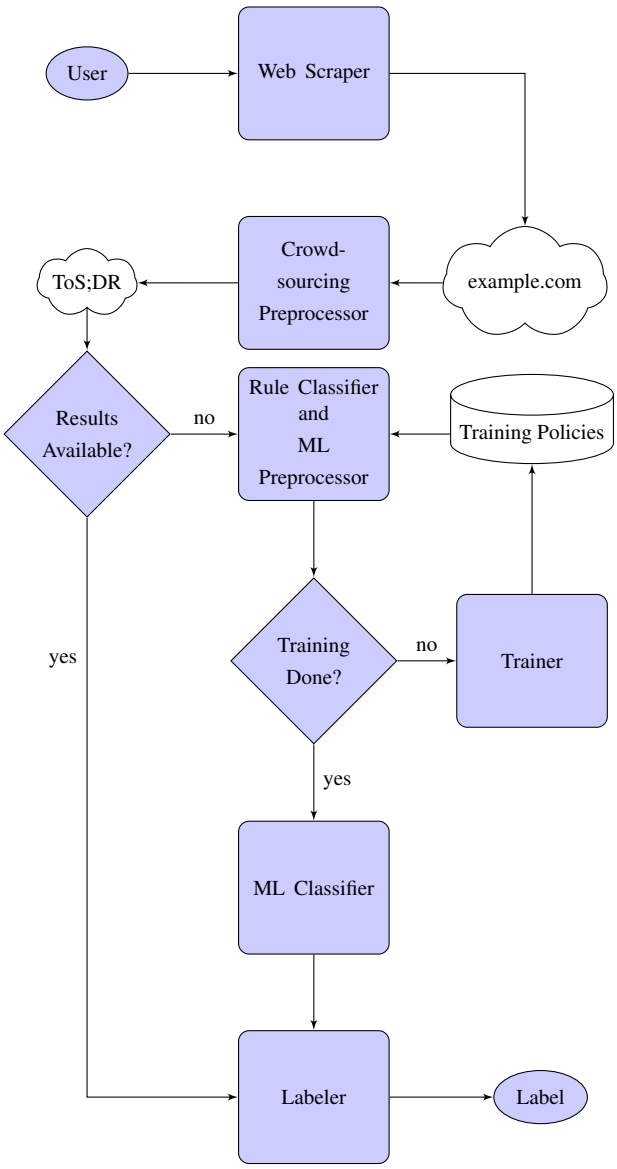


Figure 2: Simplified program flow. After the user has started the extension, the Web scraper obtains the text of the privacy policy to be analyzed (example.com) as well as the current URL (http://example.com/). The crowdsourcing preprocessor then extracts from the URL the ToS;DR identifier and checks the ToS;DR repository for results. If results are available, they are retrieved and forwarded to the labeler, which converts them to a label for display to the user. However, if no results are available on ToS;DR the policy text is analyzed. First, the rule classifier attempts a rule-based classification. However, if that is not possible the ML preprocessor prepares the ML classification. It checks if the ML classifier is already trained. If that is the case, the policy is classified by the ML classifier, assigned a label according to the classifications, and the results are displayed to the user. Otherwise, a set of training policies is analyzed by the trainer first and the program proceeds to the ML classifier and labeler afterwards. The set of training policies is included in the extension package and only needs to be analyzed for the first run of the ML classifier. Thereafter, the training results are kept in persistent storage until deletion by the user.

4 The Privee Browser Extension

We implemented Privee as a proof of concept browser extension for Google Chrome (version 35.0.1916.153). Figure 2 shows a simplified overview of the program flow. We wrote our Privee extension in JavaScript using

the jQuery library and Ajax functions for client-server communication. While we designed our extension as an end user tool, it can also be used for scientific or industrial research, for example, in order to easily compare different privacy policies to each other. In this Section we describe the various stages of program execution.

4.1 Web Scraper

The user starts the Privee extension by clicking on its icon in the Chrome toolbar. Then, the Web scraper obtains the text of the privacy policy that the user wants to analyze and retrieves the URL of the user's current website. While the rule and ML classifier analysis only works from the site that contains the policy to be analyzed, the crowdsourcing analysis works on any website whose URL contains the policy's ToS;DR identifier.

4.2 Crowdsourcing Preprocessor

The crowdsourcing preprocessor is responsible for managing the interaction with the ToS;DR repository. It receives the current URL from the Web scraper from which it extracts the ToS;Dr identifier. It then connects to the API of ToS;DR and checks for the availability of analysis results, that is, short descriptions of privacy practices and sometimes an overall letter grade. The results, if any, are forwarded to the labeler and displayed to the user. Then the extension terminates. Otherwise, the policy text, which the crowdsourcing preprocessor also received from the Web scraper, is forwarded to the rule classifier and ML preprocessor.

4.3 Rule Classifier and ML Preprocessor

Generally, classifiers can be based on rule or ML algorithms. In our preliminary experiments we found that for some classification categories a rule classifier worked better, in others an ML classifier, and in others again a combination of both [71, 76]. We will discuss our classifier selection in Section 5.1 in more detail. In this Section we will focus on the feature selection process for our rule classifier and ML preprocessor. Both rule classification and ML preprocessing are based on feature selection by means of regular expressions.

Our preliminary experiments revealed that classification performance depends strongly on feature selection. Ammar et al. [16] discuss a similar finding. Comparable to other domains [76], feature selection is particularly useful in our case for avoiding misclassifications due to the heavily imbalanced structure of privacy policies. For example, in many multi-page privacy policies there is often only one phrase that determines whether the Web service is allowed to combine the collected information with information from third parties to create personal profiles of users. Especially, supervised ML classifiers do not work well in such cases, even with undersampling (removal of uninteresting examples) or oversampling (duplication of interesting examples) [52]. Possible solutions to the problem are the separation of policies into different content zones and applying a classifier only to relevant content zones [54] or—the approach we adopted—running a classifier only on carefully selected features.

Our extension's feature selection process begins with the removal of all characters from the policy text that are not letters or whitespace and conversion of all remaining characters to lower case. However, the positions of removed punctuations are preserved because, as noted by Biagoli et al. [19], a correct analysis of the meaning of legal documents often depends on the position of punctuation. In order to identify the features that are most characteristic for a certain class we used the term frequency-inverse document frequency (tf-idf) statistic as a proxy. The tf-idf statistic measures how concentrated into relatively few documents the occurrences of a given word are in a document corpus [64]. Thus, words with high tf-idf values correlate strongly with the documents in which they appear and can be used to identify topics in that document that are not discussed in other documents. However, instead of using individual words as features we observed that the use of bigrams lead to better classification performance, which was also discussed in previous works [16, 59].

```
(ad|advertis.*) (compan.*|network.*|provider.*|
servin.*|serve.*|vendor.*)|(behav.*|context.*|
network.*|parti.*|serv.*) (ad|advertis.*)
```

Listing 1: Simplified pseudocode of the regular expression to identify whether a policy allows advertising tracking. For example, the regular expression would match "contextual advertising."

The method by which our Privee extension selects characteristic bigrams, which usually consist of two words, but can also consist of a word and a punctuation mark, is based on regular expressions. It applies a three-step process that encompasses both rule classification and ML preprocessing. To give an example, for the question whether the policy allows advertising tracking (e.g., by ad cookies) the first step consists of trying to match the regular expression in Listing 1, which identifies bigrams that nearly always indicate that advertising tracking is allowed. If any bigram in the policy matches, no further analysis happens, and the policy is classified by the rule classifier as allowing advertising tracking. If the regular expression does not match, the second step attempts to extract further features that can be associated with advertising tracking (which are, however, more gen-

eral than the previous ones). Listing 2 shows the regular expression used for the second step.

```
(ad|advertis|market) (.+)|(.+) (ad|advertis|
market)
```

Listing 2: Simplified pseudocode of the regular expression to extract relevant phrases for advertising tracking. For example, the regular expression would match "no advertising."

The second step—the ML preprocessing—is of particular importance for our analysis because it prepares classification of the most difficult cases. It extracts the features on which the ML classifier will run later. To that end, it first uses the Porter stemmer [63] to reduce words to their morphological root [19]. Such stemming has the effect that words with common semantics are clustered together [41]. For example, "collection," "collected," and "collect" are all stemmed into "collect." As a side note, while stemming had some impact, we did not find a substantial performance increase for running the ML classifier on stemmed features compared to unstemmed features. In the third step, if no features were extracted in the two previous steps, the policy is classified as not allowing advertising tracking.

4.4 Trainer

In the training stage our Privee extension checks whether the ML classifier is already trained. If that is not the case, a corpus of training policies is preprocessed and analyzed. The analysis of a training policy is similar to the analysis of a user-selected policy, except that the extension does not check for crowdsourcing results and only applies the second and third step of the rule classifier and ML preprocessor phase. The trainer's purpose is to gather statistical information about the features in the training corpus in order to prepare the classification of the user-selected policy. It stores the training results locally in the user's browser memory using persistent Web storage, which is, in principle, similar to cookie storage.

4.5 Training Data

The training policies are held in a database that is included in the extension package. The database holds a total of 100 training policies. In order to obtain a representative cross section of training policies, we selected the majority of our policies randomly from the Alexa top 500 websites for the U.S. [6] across various domains (banking, car rental, social networking, etc.). However, we also included a few random policies from lesser frequented U.S. sites and sites from other countries that published privacy policies in English. The trainer accesses these training policies one after another and adds the training results successively to the client's Web storage. After all results are added the ML classifier is ready for classification.

4.6 ML Classifier

We now describe the ML classifier design (Section 4.6.1) and the classification categories (Section 4.6.2).

4.6.1 ML Classifier Design

In order to test the suitability of different ML algorithms for analyzing privacy policies we performed preliminary experiments using the Weka library [43]. Performance for the different algorithms varied. We tested all algorithms available on Weka, among others the Sequential Minimal Optimization (SMO) algorithm with different kernels (linear, polynomial, radial basis function), random forest, J48 (C4.5), IBk nearest neighbor, and various Bayesian algorithms (Bernoulli naive Bayes, multinomial naive Bayes, Bayes Net). Surprisingly, the Bayesian algorithms were among the best performers. Therefore, we implemented naive Bayes in its Bernoulli and multinomial version. Because the multinomial version ultimately proved to have better performance, we settled on this algorithm.

As Manning et al. [56] observed, naive Bayes classifiers have good accuracy for many tasks and are very efficient, especially, for high-dimensional vectors, and they have the advantage that training and classification can be accomplished with one pass over the data. Our naive Bayes implementation is based on their specification [56]. In general, naive Bayes classifiers make use of Bayes' theorem. The probability, P, of a document, d, being in a category, c, is

$$P(c|d) \propto P(c) \prod_{1 \le k \le n_d} P(t_k|c), \tag{1}$$

where $P(c)$ is the prior probability of a document occurring in category c, n_d is the number of terms in d that are used for the classification decision, and $P(t_k|c)$ is the conditional probability of term t_k occurring in a document of category c [56]. In other words, $P(t_k|c)$ is interpreted as a measure of how much evidence t_k contributes for c being the correct category [56]. The best category to select for a document in a naive Bayes classification is the category for which it holds that

$$\arg\max_{c \in \mathbb{C}} \hat{P}(c|d) = \arg\max_{c \in \mathbb{C}} \hat{P}(c) \prod_{1 \le k \le n_d} \hat{P}(t_k|c), \tag{2}$$

where $\mathbb{C}$ is a set of categories, which, in our case, is always of size two (e.g., {ad tracking, no ad tracking}).

The naive assumption is that the probabilities of individual terms within a document are independent of each other given the category [41]. However, our implementation differs from the standard implementation and tries to alleviate the independence assumption. Instead of processing individual words of the policies we try to capture some context by processing bigrams.

Analyzing the content of a privacy policy requires multiple classification decisions. For example, the classifier has to decide whether personal information can be collected, disclosed to advertisers, retained indefinitely, and so on. This type of classification is known as multi-label classification because each analyzed document can receive more than one label. One commonly used approach for multi-label classification with L labels consists of dividing the task into $|L|$ binary classification tasks [74]. However, other solutions handle multi-label data directly by extending specific learning algorithms [74]. We found it simpler to implement the first approach. Specifically, at execution time we create multiple classifier instances—one for each classification category—by running the classifier on category-specific features extracted by the ML preprocessor.

4.6.2 Classification Categories

For which types of information should privacy policies actually be analyzed? In answering this question, one starting point are fair information practices [25]. Another one are the policies themselves. After all, while it is true that privacy law in the U.S. generally does not require policies to have a particular content, it can be observed that all policies conventionally touch upon four different themes: information collection, disclosure, use, and management (management refers to the handling of information, for example, whether information is encrypted). The four themes can be analyzed on different levels of abstraction. For example, for disclosure of information, it could simply be analyzed whether information is disclosed to outside parties in general, or it could be investigated more specifically whether information is disclosed to service providers, advertisers, governmental agencies, credit bureaus, and so on.

At this point it should be noted that not all information needs to be analyzed. In some instances privacy policies simply repeat mandatory law without creating any new rights or obligations. For example, a federal statute in the U.S.—18 U.S.C. §2703(c)(1)(A) and (B)—provides that the government can demand the disclosure of customer information from a Web service provider after obtaining a warrant or suitable court order. As this law applies independently of a privacy policy containing an explicit statement to that end, the provision that the provider will disclose information to a governmental entity under the requirements of the law can be inferred from the law itself. In fact, even if a privacy policy states to the contrary, it should be assumed that such information disclosure will occur. Furthermore, if privacy policies stay silent on certain subject matters, default rules might apply and fill the gaps.

Another good indicator of what information should be classified is provided by user studies. According to one study [30], knowing about sharing, use, and purpose of information collection is very important to 79%, 75%, and 74% of users, respectively. Similarly, in another study [11] users showed concern for the types of personal information collected, how personal information is collected, behavioral profiling, and the purposes for which the information may be used. While it was only an issue of minor interest earlier [30], the question how long a company keeps personal information about its users is a topic of increasing importance [11]. Based on these findings, we decided to perform six different binary classifications, that is, whether or not a policy

- allows collection of personal information from users (Collection);
- provides encryption for information storage or transmission (Encryption);
- allows ad tracking by means of ad cookies or other trackers (Ad Tracking);
- restricts archiving of personal information to a limited time period (Limited Retention);
- allows the aggregation of information collected from users with information from third parties (Profiling);
- allows disclosure of personal information to advertisers (Ad Disclosure).

For purposes of our analysis, where applicable, it is assumed that the user has an account with the Web service whose policy is analyzed and is participating in any offered sweepstakes or the like. Thus, for example, if a policy states that the service provider only collects personal information from registered users, the policy is analyzed from the perspective of a registered user. Also, if certain actions are dependent on the user's consent, opt-in, or opt-out, it is assumed that the user consented, opted in, or did not opt out, respectively. As it was our goal to make the analysis results intuitively comprehensible to casual users, which needs to be confirmed by user studies, we tried to avoid technical terms. In particular, the term "personal information" is identical to what is known in the privacy community as personally identifiable information (PII) (while "information" on its own also encompasses non-PII, e.g., user agent information).

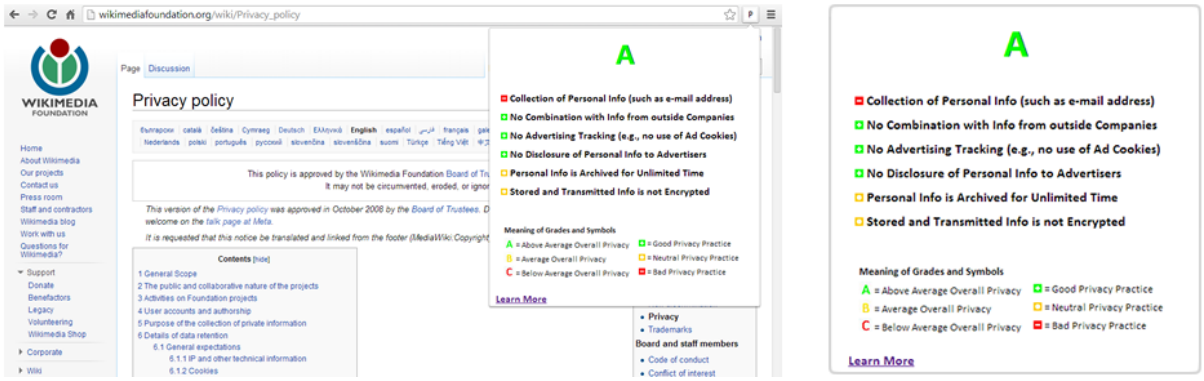


Figure 3: Privee extension screenshot and detailed label view. The result of the privacy policy analysis is shown to the user in a pop-up.

It is noteworthy that some of the analyzed criteria correspond to the semantics of the P3P Compact Specification [2]. For example, the P3P token NOI indicates that a Web service does not collect identified data while ALL means that it has access to all identified data. Thus, NOI and ALL correspond to our collection category. Also, in P3P the token IND means that information is retained for an indeterminate period of time, and, consequently, is equivalently expressed when our classifier comes to the conclusion that no limited retention exists. Further, PSA, PSD, IVA, and IVD are tokens similar to our profiling category. Generally, the correspondence between the semantics of the P3P tokens and our categories suggests that it is possible to automatically classify natural language privacy policies to obtain the same information that Web services would include in P3P policies without actually requiring them to have such.

4.7 Labeler

Our extension's labeler is responsible for creating an output label. As it was shown that users casually familiar with privacy questions were able to understand privacy policies faster and more accurately when those policies were presented in a standardized format [49] and that most users had a preference for standardized labels over full policy texts [49, 50], we created a short standardized label format. Generally, a label can be structured in one or multiple dimensions. The multidimensional approach has the advantage that it can succinctly display different privacy practices for different types of information. However, we chose a one-dimensional format as such were shown to be substantially more comprehensible [51, 66].

In addition to the descriptions for the classifications, the labeler also labels each policy with an overall letter grade, which depends on the classifications. More specifically, the grade is determined by the number of points, p, a policy is assigned. For collection, profiling, ad tracking, and ad disclosure a policy receives one minus point, respectively. However, for not allowing one of these practices a policy receives one plus point. However, a policy receives a plus point for featuring limited retention or encryption, respectively. As most policies in the training set had zero points, we took zero points as a mean and assigned grades as follows:

- A (above average overall privacy) if $p > 1$;
- B (average overall privacy) if $1 \leq p \geq -1$;
- C (below average overall privacy) if $p < -1$.

After the points are assigned to a policy, the corresponding label is displayed to the user as shown in Figure 3. As we intended to avoid confusion about the meaning of icons [45], we used short descriptions instead. The text in the pop-up is animated. If the user moves the mouse over it, further information is provided. The user can also find more detailed explanations about the categories and the grading by clicking on the blue "Learn More" link at the bottom of the label. It should be noted that analysis results retrieved from ToS;DR usually differ in content from our classification results, and are, consequently, displayed in a different label format.

5 Experimental Results

For our experiments we ran our Privee extension on a test set of 50 policies. Before this test phase we trained the ML classifier (with the 100 training policies that are included in the extension package) and tuned it (with a validation set of 50 policies). During the training, validation, and test phases we disabled the retrieval of crowdsourcing results. Consequently, our experimental results only refer to rule and ML classification. The policies of the test and validation sets were selected according to the same criteria as described for the training set in Section

	Base.	Acc.	Prec.	Rec.	F-1
Overall	**68%**	**84%**	**94%**	**89%**	**90%**
Collection	100%	100%	100%	100%	100%
Encryption	52%	98%	96%	100%	98%
Ad Tracking	64%	96%	94%	100%	97%
L. Retention	74%	90%	83%	77%	80%
Profiling	52%	86%	100%	71%	83%
Ad Disclosure	66%	76%	69%	53%	60%

Table 1: Privee extension performance overall and per category. For the 300 test classifications (six classifications for each of the 50 test policies) we observed 27 misclassifications. 154 classifications were made by the rule classifier and 146 by the ML classifier. The rule classifier had 11 misclassifications (2 false positives and 9 false negatives) and the ML classifier had 16 misclassifications (7 false positives and 9 false negatives). It may be possible to decrease the number of false negatives by adding more rules and training examples. For the ad tracking category the rule classifier had an F-1 score of 98% and the ML classifier had an F-1 score of 94%. For the profiling category the rule classifier had an F-1 score of 100% and the ML classifier had an F-1 score of 53%. 28% of the policies received a grade of A, 50% a B, and 22% a C.

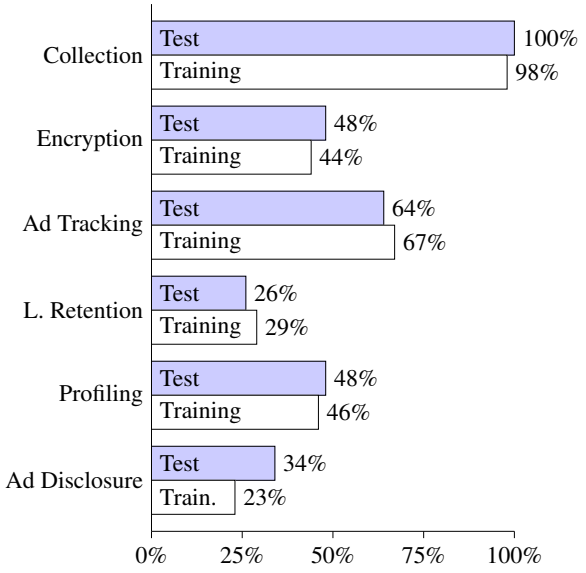


Figure 4: Annotation of positive cases in percent for the 50 test policies (blue) and the 100 training policies (white).

4.5. In this Section we first discuss the classification performance (Section 5.1), then the gold standard that we used to measure the performance (Section 5.2), and finally the computational performance (Section 5.3).

5.1 Classification Performance

In the validation phase we experimented with different classifier configurations for each of our six classification tasks. For the ad tracking and profiling categories the combination of the rule and ML classifier lead to the best results. However, for collection, limited retention, and ad disclosure the ML classifier on its own was preferable. Conversely, for the encryption category the rule classifier on its own was the best. It seems that the language used for describing encryption practices is often very specific making the rule classifier the first choice. Words such as "ssl" are very distinctive identifiers for encryption provisions. Other categories use more general language that could be used in many contexts. For example, phrases related to time periods must not necessarily refer to limited retention. For those instances the ML classifier seems to perform better. However, if categories exhibit both specific and general language the combination of the rule and ML classifier is preferable.

The results of our extension's privacy policy analysis are based on the processing of natural language. However, as natural language is often subject to different interpretations, the question becomes how the results can be verified in a meaningful way. Commonly applied metrics for verifying natural language classification tasks are accuracy (Acc.), precision (Prec.), recall (Rec.), and F-1 score (F-1). Accuracy is the fraction of classifications that are correct [56]. Precision is the fraction of retrieved documents that are relevant, and recall is the fraction of relevant documents that are retrieved [56]. Precision and recall are often combined in their harmonic mean, known as the F-1 score [46].

In order to analyze our extension's performance we calculated the accuracy, precision, recall, and F-1 score for the test policy set classifications. Table 1 shows the overall performance and the performance for each classification category. We also calculated the baseline accuracy (Base.) for comparison against the actual accuracy. The baseline accuracy for each category was determined by always selecting the classification corresponding to the annotation that occurred the most in the training set annotations, which we report in Figure 4. The baseline accuracy for the overall performance is the mean of the category baseline accuracies. Because the classification of privacy policies is a multi-label classification task, as described in Section 4.6.1, we calculated the overall results based on the method for measuring multi-label classifications given by Godbole and Sarawagi [42]. According to their method, for each document, d_j in set D, let t_j be the true set of labels and s_j be the predicted set of labels. Then we obtain the means by

$$Acc(D) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|t_j \cap s_j|}{|t_j \cup s_j|}, \tag{3}$$

$$Prec(D) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|t_j \cap s_j|}{|s_j|}, \tag{4}$$

$$Rec(D) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|t_j \cap s_j|}{|t_j|}, \tag{5}$$

$$F\text{-}1(D) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{2\,Prec(d_j)\,Rec(d_j)}{(Prec(d_j)+Rec(d_j))}. \quad (6)$$

From Table 1 it can be observed that the accuracies are at least as good as the corresponding baseline accuracies. For example, in the case of limited retention the baseline classifies all policies as not providing for limited retention because, as show in Figure 4, only 29% of the training policies were annotated as having a limited retention period, which would lead to a less accurate classification of 74% in the test set compared to the actual accuracy of 90%. For the collection category it should be noted that there is a strong bias because nearly every policy allows the collection of personal information. However, in our validation set we had two policies that did not allow this practice, but still were correctly classified by our extension. Generally, our F-1 performance results fall squarely within the range reported in the earlier works. For identifying law enforcement disclosures Ammar et Al. [16] achieved an F-1 score of 76% and Costante et al. reported a score of 83% for recognizing types of collected information [26] and 92% for identifying topics discussed in privacy policies [27].

In order to investigate the reasons behind our extension's performance we used two binary logistic regression models. Binary logistic regression is a statistical method for evaluating the dependence of a binary variable (the dependent variable) on one or more other variables (the independent variable(s)). In our first model each of the 50 test policies was represented by one data point with the dependent variable identifying whether it had any misclassification and the independent variables identifying (1) the policy's length in words, (2) its mean Semantic Diversity (SemD) value [44], and (3) whether there was any disagreement among the annotators in annotating the policy (Disag.). In our second model we represented each of 185 individual test classifications by one data point with the dependent variable identifying whether it was a misclassification and the independent variables identifying (1) the length (in words) of the text that the rule classifier or ML preprocessor extracted for the classification, (2) the text's mean SemD value, and (3) whether there was annotator disagreement on the annotation corresponding to the classification.

Hoffman et al.'s [44] SemD value is an ambiguity measure for words based on latent semantic analysis, that is, the similarity of contexts in which words are used. It can range from 0 (highly unambiguous) to 2.5 (highly ambiguous). We represented the semantic diversity of a document (i.e., a policy or extracted text) by the mean SemD value of its words. However, as Hoffman et al. only provide SemD values for words on which they had sufficient analytical data (31,739 different words in total), some words could not be taken into account for calculating a document's mean SemD value. Thus, in order to avoid skewing of mean SemD values in our models, we only considered documents that had SemD values for at least 80% of their words. In our first model all test policies were above this threshold. However, in our second model we excluded some of the 300 classifications. Particularly, all encryption classifications were excluded because words, such as "encryption" and "ssl" occurred often and had no SemD value. Also, in the second model the mean SemD value of an extracted text was calculated after stemming its words with the Porter stemmer and obtaining the SemD values for the resulting word stems (while the SemD value of each word stem was calculated from the mean SemD value of all words that have the respective word stem).

Per Policy	Length	SemD	Disag.
Mean	2873.4	2.08	0.6
Significance (P)	0.64	0.74	0.34
Odds Ratio (Z)	1.15	1.11	0.54
95% Confidence Interval (Z)	0.64-2.08	0.61-2.01	0.16-1.89

Table 2: Results of the first logistic regression model. The Nagelkerke pseudo R^2 is 0.03 and the Hosmer and Lemeshow value 0.13.

Per Extr. Text	Length	SemD	Disag.
Mean	37.38	1.87	0.17
Significance (P)	0.22	**0.02**	0.81
Odds Ratio (Z)	0.58	**2.07**	0.86
95% Confidence Interval (Z)	0.24-1.38	1.12-3.81	0.25-2.97

Table 3: Results of the second logistic regression model. The Nagelkerke pseudo R^2 is 0.11 and the Hosmer and Lemeshow value 0.051.

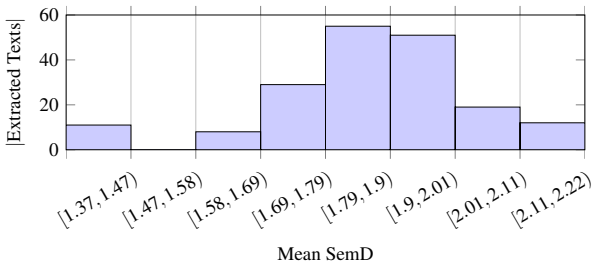


Figure 5: Mean SemD value distribution for the 185 extracted texts. The standard deviation is 0.17.

For our first model the results of our analysis are shown in Table 2 and for our second model in Table 3. Figure 5 shows the distribution of mean SemD values for the extracted texts in our second model. Using the

Wald test, we evaluated the relationship between an independent variable and the dependent variable through the P value relating to the coefficient of that independent variable. If the P value is less than 0.05, we reject the null hypothesis, i.e., that that coefficient is zero. Looking at our results, it is noteworthy that both models do not reveal a statistically relevant correlation between the annotator disagreements and misclassifications. Thus, a document with a disagreement did not have a higher likelihood of being misclassified than one without. However, it is striking that the second model has a P value of 0.02 for the SemD variable. Standardizing our data points into Z scores and calculating the odds ratios it becomes clear that an increase of the mean SemD value in an extracted text by 0.17 (one standard deviation) increased the likelihood of a misclassification by 2.07 times (odds ratio). Consequently, our second model shows that the ambiguity of text in privacy policies, as measured by semantic diversity, has statistical significance for whether a classification decision is more likely to succeed or fail.

Besides evaluating the statistical significance of individual variables, we also assessed the overall model fit. While the goodness of fit of linear regression models is usually evaluated based on the R^2 value, which measures the square of the sample correlation coefficient between the actual values of the dependent variable and the predicted values (in other words, the R^2 value can be understood as the proportion of the variance in a dependent variable attributable to the variance in the independent variable), there is no consensus for measuring the fit of binary logistic regression models. Various pseudo R^2 metrics are discussed. We used the Nagelkerke pseudo R^2 because it can range from 0 to 1 allowing an easy comparison to the regular R^2 (which, however, has to account for the fact that the Nagelkerke pseudo R^2 is often substantially lower than the regular R^2). While the Nagelkerke pseudo R^2 of 0.03 for our first model indicates a poor fit, the value of 0.11 for our second model can be interpreted as moderate. Further, the Hosmer and Lemeshow test, whose values were over 0.05 for both of our models, demonstrates the model fit as well.

In addition to the experiments just discussed, we also evaluated our models with further independent variables. Specifically, we evaluated our first model with the policy publication year, the second model with the extracted texts' mean tf-idf values, and both models with Flesch-Kincaid readability scores as independent variables. Also, using only ML classifications we evaluated our second model with the number of available training examples as independent variable. Only for the latter we found statistical significance at the 0.05 level. The number of training examples correlated to ML classification performance, which confirms Ammar et al.'s respective conjecture [16]. The more training examples the ML classifier had, the less likely a misclassification became.

5.2 Inter-annotator Agreement

Having discussed the classification performance, we now turn to the gold standard that we used to measure that performance. For our performance results to be reliable our gold standard must be reliable. One way of producing a gold standard for privacy policies is to ask the providers whose policies are analyzed to explain their meaning [11]. However, this approach should not be used, at least in the U.S., because the Restatement of Contracts provides that a contract term is generally given the meaning that *all* parties associate with it (Restatement (Second) of Contracts, §201). Consequently, policies should be interpreted from the perspective of both the provider and user. The interpretation would evaluate whether their perspectives lead to identical meanings or, if that is not the case, which one should prevail under applicable principles of legal interpretation. In addition, since technical terms are generally given technical meaning (Restatement (Second) of Contracts, §202(3)(b)), it would be advantageous if the interpretation is performed by annotators familiar with the terminology commonly used in privacy policies. The higher the number of annotations on which the annotators agree, that is, the higher the inter-annotator agreement, the more reliable the gold standard will be.

Because the annotation of a large number of documents can be very laborious, it is sufficient under current best practices for producing a gold standard to measure inter-annotator agreement only on a data sample [62], such that it can be inferred that the annotation of the remainder documents is reliable as well. Following this practice, we only measured the inter-annotator agreement for our test set, which would then provide an indicator for the reliability of our training and validation set annotation as well. To that end, one author annotated all policies and additional annotations were obtained for the test policies from two other annotators. All annotators worked independently from each other. As the author who annotated the policies studied law and has expertise in privacy law and the two other annotators were law students with training in privacy law, all annotators were considered equally qualified, and the annotations for the gold standard were selected according to majority vote (i.e., at least two annotators agreed). After the annotations of the test policies were made, we ran our extension on these policies and compared its classifications to the annotations, which gave us the results in Table 1.

The reliability of our gold standard depends on the degree to which the annotators agreed on the annotations. There are various measures for inter-annotator agreement. One basic measure is the count of disagreements.

	Disag.	% Ag.	K.'s α/F.'s κ
Overall	**8.12**	**84%**	**0.77**
Collection	0	100%	1
Encryption	6	88%	0.84
Ad Tracking	7	86%	0.8
L. Retention	9	82%	0.68
Profiling	11	78%	0.71
Ad Disclosure	16	68%	0.56

Table 4: Inter-annotator agreement for the 50 test policies. The values for Krippendorff's α and Fleiss' κ are identical.

Per Policy	Length	SemD	Flesch-K.
Mean	2873.4	2.08	14.53
Significance (P)	0.2	0.11	0.76
Odds Ratio (Z)	1.65	1.87	1.12
95% Confidence Interval (Z)	0.78-3.52	0.87-4	0.55-2.29

Table 5: Results of the third logistic regression model. The Nagelkerke pseudo R^2 is 0.19 and the Hosmer and Lemeshow value 0.52.

Per Section	Length	SemD	Flesch-K.
Mean	306.76	2.08	15.59
Significance (P)	0.29	**0.04**	0.49
Odds Ratio (Z)	1.18	**1.51**	0.86
95% Confidence Interval (Z)	0.87-1.6	1.02-2.22	0.56-1.32

Table 6: Results of the fourth logistic regression model. The Nagelkerke pseudo R^2 is 0.05 and the Hosmer and Lemeshow value 0.83.

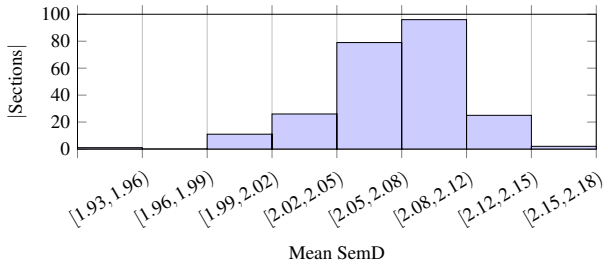


Figure 6: Mean SemD value distribution for the 240 policy sections. The standard deviation is 0.03.

Another one is the percentage of agreement (% Ag.), which is the fraction of documents on which the annotators agree [17]. However, disagreement count and percentage of agreement have the disadvantage that they do not account for chance agreement. In this regard, chance-corrected measures, such as Krippendorff's α (K.'s α) [53] and Fleiss' κ (F.'s κ) [40] are superior. For Krippendorff's α and Fleiss' κ the possible values are constrained to the interval $[-1;1]$, where 1 means perfect agreement, -1 means perfect disagreement, and 0 means that agreement is equal to chance [37]. Generally, values above 0.8 are considered as good agreement, values between 0.67 and 0.8 as fair agreement, and values below 0.67 as dubious [56]. However, those ranges are only guidelines [17]. Particularly, ML algorithms can tolerate data with lower reliability as long as the disagreement looks like random noise [68].

Based on the best practices and guidelines for interpreting inter-annotator agreement measurements, our results in Table 4 confirm the general reliability of our annotations and, consequently, of our gold standard. For every individual category, except for the ad disclosure category, we obtained Krippendorff's α values indicating fair or good agreement. In addition, the overall mean agreement across categories is 0.77, and, therefore, provides evidence for fair overall agreement as well. For the overall agreement it should be noted that, corresponding to the multi-label classification task, the annotation of privacy policies is a multi-label annotation task as well. However, there are only very few multi-label annotation metrics, such as Passonneau's Measuring Agreement on Set-valued Items (MASI) [61]. As none of the metrics were suitable for our purposes, we selected as overall metric the mean over the results of the individual classification categories.

We investigated our inter-annotator agreement results by applying a third and fourth binary logistic regression model. In our third model each of the 50 test policies was represented by one data point with the dependent variable identifying whether the annotators had any disagreement in annotating the policy and the independent variables identifying (1) the policy's length in words, (2) its mean SemD value, and (3) its Flesch-Kincaid score. In our fourth model we represented each of 240 individual annotations by one data point with the dependent variable identifying whether the annotators disagreed for that annotation and the independent variables identifying (1) the length (in words) of the policy text section that the annotation is referring to, (2) the section's mean SemD value, and (3) its Flesch-Kincaid score. For the fourth model we excluded some of the 300 annotations because not every policy had a section for each category. For example, some policies did not discuss advertisement or disclosure of information. The Flesch-Kincaid readability score measures the number of school years an average reader would need to understand a text.

For our third and fourth model the results of our analysis are shown in Table 5 and 6, respectively. Figure 6 shows the distribution of mean SemD values for the policy sections in our fourth model. Both models were

significant, as indicated by their Nagelkerke and Hosmer and Lemeshow values. Our results confirm that the readability of policies, as measured by the Flesch-Kincaid score, does not impact their comprehensibility [58]. In our third model we were unable to identify any statistically relevant variables (although, semantic diversity and length may be statistically significant in a larger data set). However, our fourth model proved to be more meaningful. Remarkably, corresponding to our finding in Section 5.1, according to which classifier performance correlates to semantic diversity, the statistically relevant P value of 0.04 for the mean SemD variable also indicates a correlation of inter-annotator agreement to semantic diversity. Standardizing our data points into Z scores and calculating the odds ratios it becomes clear that an increase of the mean SemD value of a section by 0.03 (one standard deviation) increased the likelihood of a disagreement by 1.51 times (odds ratio). It is astounding that even qualified annotators trained in privacy law had difficulties to avoid disagreements when semantic diversity increased to slightly above-mean levels.

While neither our first nor our second model in Section 5.1 showed a correlation between inter-annotator agreement and classifier performance, the results for our second and fourth model demonstrate that performance and agreement both correlate to one common variable—semantic diversity. More specifically, performance correlates to the semantic diversity of extracted text phrases and agreement correlates to the semantic diversity of policy sections. This result suggests, for example, that the relatively high number of misclassifications and disagreements in the ad disclosure category is inherent in the nature of the category. Indeed, in cases of fuzzy categories disagreements among annotators do not necessarily reflect a quality problem of the gold standard, but rather a structural property of the annotation task, which can serve as an important source of empirical information about the structural properties of the investigated category [13]. Thus, it is no surprise that for all six categories the values of Krippendorff's α correlate to the F-1 scores. The higher the value of Krippendorff's α, the higher the F-1 score. Figure 7 shows the correlation.

As both classifier performance and inter-annotator agreement decrease with an increase in semantic diversity, the practicability of the notice-and-choice principle becomes questionable. After all, privacy policies can only provide adequate notice (and choice) if they are not too ambiguous. In order to further examine policy ambiguity we calculated the mean SemD value for our test policies over time. Our test set analysis exhibited a statistically significant trend of decreasing semantic diversity with a P value of 0.049. Figure 8 illustrates our approach. We can think of two explanations for the decrease over time. First, it could be a consequence of the FTC's enforcement actions and its call for policies to "be clearer, shorter, and more standardized" [38]. Second, we might be in the midst of a consolidation process leading to more standardized policy language. As de Maat et al [34] observed, drafters of legal documents tend to use language that adheres to writing conventions of earlier texts and similar statements. Independent of the reason, our result suggests that the notice-and-choice principle can overcome the problem of ambiguity over time.

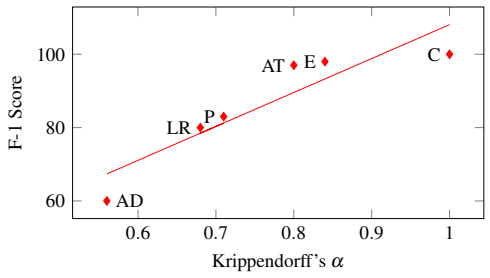


Figure 7: Linear regression plot with the F-1 score as dependent variable and Krippendorff's α as independent variable. The coordinate labels identify the categories: AD = Ad Disclosure, LR = Limited Retention, P = Profiling, AT = Ad Tracking, E = Encryption, and C = Collection. With an R^2 value of 0.83 the model has an excellent fit, which, however, should be interpreted in light of the small number of data points.

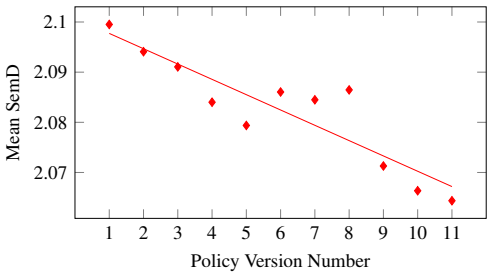


Figure 8: Linear regression plot for Symantec's privacy policy (which was part of our test set) with the mean SemD value of a policy version as dependent variable and the policy version number as independent variable. The first version of Symantec's policy [73] dates back to August 5, 1999, and the eleventh version [72] was adopted on August 12, 2013. The mean SemD value of Symantec's privacy policy decreased from 2.1 in the first version to 2.06 in the eleventh version as shown. We observed a similar decrease for 29 out of 44 test policies (6 of the test policies were only available in a single version and, therefore, could not be included in our analysis. However, for the 44 included policies we obtained on average 8 different versions over time.).

5.3 Computational Performance

We finish the discussion of our experimental results with our extension's computational performance. We report the mean duration in seconds for obtaining analysis results for each of 50 randomly selected policies from ToS;DR (Crowdsourcing), processing each of the 50 test policies (Classifier), and processing each of the 50 test

Per Policy	Crowdsourcing	Classifier	Training
Mean	0.39 sec	0.78 sec	20.29 sec

Table 7: Computational performance of the Privee extension. The performance was evaluated on a Windows laptop with Intel Core2 Duo CPU at 2.13 GHz with 4 GB RAM. The space requirements for the installation on the hard disk are 2.11 MB (including 1.7 MB of training data and 286 KB for the jQuery library) and additional 230 KB during the program execution for storing training results.

policies each with initial training (Training) in Table 7. Notably, retrieving policy results from ToS;DR is twice as fast as analyzing a policy with our classifiers.

6 Conclusion

We introduced Privee—a novel concept for analyzing natural language privacy policies based on crowdsourcing and automatic classification techniques. We implemented Privee in a proof of concept browser extension for Google Chrome, and our automatic classifiers achieved an overall F-1 score of 90%. Our experimental results revealed that the automatic classification of privacy policies encounters the same constraint as human interpretation—the ambiguity of natural language, as measured by semantic diversity. Such ambiguity seems to present an inherent limitation of what automatic privacy policy analysis can accomplish. Thus, on a more fundamental level, the viability of the notice-and-choice principle might be called into question altogether. However, based on the decrease of policy ambiguity over time we would caution to draw such conclusion. We remain optimistic that the current notice-and-choice ecosystem is workable and can be successfully supplemented by Privee.

The most important task for making the notice-and-choice principle work is to decrease policy ambiguity. However, other areas require work as well: What are the types of information that policies should be analyzed for? What is the most usable design? What are the best features and algorithms? Are more intricate ML or natural language processing algorithms better at resolving ambiguities? What is the ideal size and composition of the training set? How can the interaction between the classifier and crowdsourcing analysis be improved? In particular, how can a program connect to many crowdsourcing repositories, and, possibly, decide which analysis is the best? Can crowdsourced policy results be used by the classifiers as training data? How can it be assured that the crowdsourcing results are always up to date? How can the quality and consistency of crowdsourcing and ML analyses be guaranteed? And, finally, what solutions are viable for different legal systems and the mobile world?

7 Acknowledgments

For their advice and help we thank Kapil Thadani, Rebecca Passonneau, Florencia Marotta-Wurgler, Sydney Archibald, Shaina Hyder, Jie S. Li, Gonzalo Mena, Paul Schwartz, Kathleen McKeown, Tony Jebara, Jeroen Geertzen, Michele Nelson, and the commentators at the 7th Annual Privacy Law Scholars Conference (2014) and the Workshop on the Future of Privacy Notice and Choice (2014). We also thank the anonymous reviewers for their valuable feedback.

8 Availability

Our Privee extension is available at `http://www.sebastianzimmeck.de/publications.html`.

References

[1] Claridge v. RockYou, Inc., 785 F. Supp. 2d 855 (N.D. Cal. 2011).

[2] P3P compact policy cross-reference. `http://compactprivacypolicy.org/compact_token_reference.htm`. Last accessed: July 1, 2014.

[3] Privacy Icons. `http://www.azarask.in/blog/post/privacy-icons/`. Last accessed: July 1, 2014.

[4] privacychoice. `http://www.privacychoice.org`. Last accessed: July 1, 2014.

[5] Terms of Service; Didn't Read (ToS;DR). `http://tosdr.org/index.html`. Last accessed: July 1, 2014.

[6] Top sites in United States. `http://www.alexa.com/topsites/countries/US`. Last accessed: July 1, 2014.

[7] TOSBack. `http://tosback.org/`. Last accessed: July 1, 2014.

[8] TOSBack2. `https://github.com/pde/tosback2`. Last accessed: July 1, 2014.

[9] Usable Privacy Policy Project. `http://www.usableprivacy.org/home`. Last accessed: July 1, 2014.

[10] Platform for Internet Content Selection (PICS). `http://www.w3.org/PICS/`, 1997. Last accessed: July 1, 2014.

[11] KnowPrivacy. `http://knowprivacy.org/`, June 2009. Last accessed: July 1, 2014.

[12] eXtensible Access Control Markup Language (XACML) version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`, Jan. 2013. Last accessed: July 1, 2014.

[13] ADREEVSKAIA, A., AND BERGLER, S. Mining WordNet for fuzzy sentiment: Sentiment tag extraction from WordNet glosses. In *11th conference of the European chapter of the Association for Computational Linguistics* (Stroudsburg, PA, USA, 2006), EACL '06, ACL, pp. 209–216.

[14] AGRAWAL, R., KIERNAN, J., SRIKANT, R., AND XU, Y. An XPath-based preference language for P3P. In *Proceedings of the 12th international conference on World Wide Web* (New York, NY, USA, 2003), WWW '03, ACM, pp. 629–639.

[15] AÏMEUR, E., GAMBS, S., AND HO, A. UPP: User privacy policy for social networking sites. In *Fourth International Conference on Internet and Web applications and services* (Washington, DC, USA, 2009), ICIW '09, IEEE Computer Society, pp. 267–272.

[16] AMMAR, W., WILSON, S., SADEH, N., AND SMITH, N. Automatic categorization of privacy policies: A pilot study. Tech. Rep. CMU-ISR-12-114, CMU-LTI-12-019, Carnegie Mellon University, Dec. 2012.

[17] ARTSTEIN, R., AND POESIO, M. Inter-coder agreement for computational linguistics. *Comput. Linguist. 34*, 4 (Dec. 2008), 555–596.

[18] ASHLEY, P., HADA, S., KARJOTH, G., POWERS, C., AND SCHUNTER, M. Enterprise Privacy Authorization Language (EPAL 1.2). Tech. rep., IBM, Nov. 2003.

[19] BIAGIOLI, C., FRANCESCONI, E., PASSERINI, A., MONTEMAGNI, S., AND SORIA, C. Automatic semantics extraction in law documents. In *Proceedings of the 10th International Conference on Artificial Intelligence and Law* (New York, NY, USA, 2005), ICAIL '05, ACM, pp. 133–140.

[20] BREAUX, T. D., AND ANTÓN, A. I. Mining rule semantics to understand legislative compliance. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2005), WPES '05, ACM, pp. 51–54.

[21] BREAUX, T. D., AND ANTÓN, A. I. Analyzing regulatory rules for privacy and security requirements. *IEEE Trans. Software Eng. 34*, 1 (Jan. 2008), 5–20.

[22] BRODIE, C., KARAT, C.-M., KARAT, J., AND FENG, J. Usable security and privacy: a case study of developing privacy management tools. In *Proceedings of the 2005 Symposium On Usable Privacy and Security* (New York, NY, USA, 2005), SOUPS '05, ACM, pp. 35–43.

[23] BRODIE, C. A., KARAT, C.-M., AND KARAT, J. An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench. In *Proceedings of the second Symposium On Usable Privacy and Security* (New York, NY, USA, 2006), SOUPS '06, ACM, pp. 8–19.

[24] BYERS, S., CRANOR, L. F., KORMANN, D., AND MCDANIEL, P. Searching for privacy: design and implementation of a P3P-enabled search engine. In *Proceedings of the 4th international conference on Privacy Enhancing Technologies* (Berlin, Heidelberg, Germany, 2005), PET '04, Springer, pp. 314–328.

[25] CIOCCHETTI, C. A. The future of privacy policies: A privacy nutrition label filled with fair information practices. *J. Marshall J. Computer & Info. L. 26* (2008), 1–46.

[26] COSTANTE, E., DEN HARTOG, J., AND PETKOVIC, M. What websites know about you. In *DPM/SETOP* (Berlin, Heidelberg, Germany, 2012), R. D. Pietro, J. Herranz, E. Damiani, and R. State, Eds., vol. 7731 of *Lecture Notes in Computer Science*, Springer, pp. 146–159.

[27] COSTANTE, E., SUN, Y., PETKOVIĆ, M., AND DEN HARTOG, J. A machine learning solution to assess privacy policy completeness: (short paper). In *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2012), WPES '12, ACM, pp. 91–96.

[28] CRANOR, L. F. Necessary but not sufficient: Standardized mechanisms for privacy notice and choice. *J. on Telecomm. and High Tech. L. 10*, 2 (2012), 273–307.

[29] CRANOR, L. F., DOBBS, B., EGELMAN, S., HOGBEN, G., HUMPHREY, J., LANGHEINRICH, M., MARCHIORI, M., PRESLER-MARSHALL, M., REAGLE, J. M., SCHUNTER, M., STAMPLEY, D. A., AND WENNING, R. The Platform for Privacy Preferences 1.1 (P3P1.1) specification. World Wide Web Consortium, Note NOTE-P3P11-20061113, November 2006.

[30] CRANOR, L. F., GUDURU, P., AND ARJULA, M. User interfaces for privacy agents. *ACM Trans. Comput.-Hum. Interact. 13*, 2 (June 2006), 135–178.

[31] CRANOR, L. F., LANGHEINRICH, M., AND MARCHIORI, M. A P3P Preference Exchange Language 1.0 (APPEL 1.0). World Wide Web Consortium, Working Draft WD-P3P-preferences-20020415, April 2002.

[32] CRANOR, L. F., LANGHEINRICH, M., MARCHIORI, M., PRESLER-MARSHALL, M., AND REAGLE, J. M. The Platform for Privacy Preferences 1.0 (P3P1.0) specification. World Wide Web Consortium, Recommendation REC-P3P-20020416, April 2002.

[33] CRANOR, L. F., AND REIDENBERG, J. R. Can user agents accurately represent privacy notices? *TPRC* (Sept. 2002).

[34] DE MAAT, E., KRABBEN, K., AND WINKELS, R. Machine learning versus knowledge based classification of legal texts. In *Proceedings of the 2010 conference on Legal Knowledge and Information Systems: JURIX 2010: The Twenty-Third Annual Conference* (Amsterdam, The Netherlands, The Netherlands, 2010), IOS Press, pp. 87–96.

[35] DE MAAT, E., AND WINKELS, R. Automatic classification of sentences in dutch laws. In *Proceedings of the 2008 conference on Legal Knowledge and Information Systems: JURIX 2008: The Twenty-First Annual Conference* (Amsterdam, The Netherlands, The Netherlands, 2008), IOS Press, pp. 207–216.

[36] DE MAAT, E., AND WINKELS, R. A next step towards automated modelling of sources of law. In *Proceedings of the 12th International Conference on Artificial Intelligence and Law* (New York, NY, USA, 2009), ICAIL '09, ACM, pp. 31–39.

[37] DI EUGENIO, B., AND GLASS, M. The kappa statistic: a second look. *Comput. Linguist. 30*, 1 (Mar. 2004), 95–101.

[38] FEDERAL TRADE COMMISSION. Protecting consumer privacy in an era of rapid change. `http://www.ftc.gov/reports/protecting-consumer-privacy-era-rapid-change-recommendations-businesses-policymakers`, Mar. 2012. Last accessed: July 1, 2014.

[39] FISCHER-HÜBNER, S., AND ZWINGELBERG, H. UI prototypes: Policy administration and presentation - version 2. Tech. Rep. D4.3.2, Karlstad University, 2010.

[40] FLEISS, J. Measuring nominal scale agreement among many raters. *Psychological Bulletin 76*, 5 (1971), 378–382.

[41] FRANCESCONI, E., AND PASSERINI, A. Automatic classification of provisions in legislative texts. *Artif. Intell. Law 15*, 1 (Mar. 2007), 1–17.

[42] GODBOLE, S., AND SARAWAGI, S. Discriminative methods for multi-labeled classification. In *Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining* (Berlin, Heidelberg, Germany, 2004), Springer, pp. 22–30.

[43] HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. The WEKA data mining software: An update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[44] HOFFMAN, P., RALPH, M. L., AND ROGERS, T. Semantic diversity: A measure of semantic ambiguity based on variability in the contextual usage of words. *BRM 45*, 3 (2013), 718–730.

[45] HOLTZ, L.-E., NOCUN, K., AND HANSEN, M. Towards displaying privacy information with icons. In *Privacy and Identity Management for Life* (Berlin, Heidelberg, Germany, 2011), S. Fischer Hübner, P. Duquenoy, M. Hansen, R. Leenes, and G. Zhang, Eds., vol. 352 of *IFIP Advances in Information and Communication Technology*, Springer, pp. 338–348.

[46] HRIPCSAK, G., AND ROTHSCHILD, A. S. Technical brief: Agreement, the F-measure, and reliability in information retrieval. *JAMIA 12*, 3 (2005), 296–298.

[47] KARJOTH, G., SCHUNTER, M., AND WAIDNER, M. Platform for Enterprise Privacy Practices: privacy-enabled management of customer data. In *Proceedings of the 2nd international conference on Privacy Enhancing Technologies* (Berlin, Heidelberg, Germany, 2003), PET '02, Springer, pp. 69–84.

[48] KELLEY, P. G. Designing a privacy label: assisting consumer understanding of online privacy practices. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2009), CHI EA '09, ACM, pp. 3347–3352.

[49] KELLEY, P. G., BRESEE, J., CRANOR, L. F., AND REEDER, R. W. A "nutrition label" for privacy. In *Proceedings of the 5th Symposium On Usable Privacy and Security* (New York, NY, USA, 2009), SOUPS '09, ACM, pp. 4:1–4:12.

[50] KELLEY, P. G., CESCA, L., BRESEE, J., AND CRANOR, L. F. Standardizing privacy notices: an online study of the nutrition label approach. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI '10, ACM, pp. 1573–1582.

[51] KELLEY, P. G., CRANOR, L. F., AND SADEH, N. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 3393–3402.

[52] KOBYLIŃSKI, L., AND PRZEPIÓRKOWSKI, A. Definition extraction with balanced random forests. In *Proceedings of the 6th international conference on Advances in Natural Language Processing* (Berlin, Heidelberg, Germany, 2008), GoTAL '08, Springer, pp. 237–247.

[53] KRIPPENDORFF, K. *Content analysis: An introduction to its methodology.* SAGE, Beverly Hills, CA, USA, 1980.

[54] KUHN, F. A description language for content zones of German court decisions. In *Proceedings of the LREC 2010 Workshop on the Semantic Processing of Legal Texts* (2010), SPLeT '10, pp. 1–7.

[55] LEON, P. G., CRANOR, L. F., MCDONALD, A. M., AND MCGUIRE, R. Token attempt: The misrepresentation of website privacy policies through the misuse of P3P compact policy tokens. In *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2010), WPES '10, ACM, pp. 93–104.

[56] MANNING, C. D., RAGHAVAN, P., AND SCHÜTZE, H. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[57] MAROTTA-WURGLER, F. Does contract disclosure matter? *JITE 168*, 1 (2012), 94–119.

[58] MCDONALD, A. M., REEDER, R. W., KELLEY, P. G., AND CRANOR, L. F. A comparative study of online privacy policies and formats. In *Proceedings of the 9th international symposium on Privacy Enhancing Technologies* (Berlin, Heidelberg, Germany, 2009), PETS '09, Springer, pp. 37–55.

[59] MOENS, M.-F., BOIY, E., PALAU, R. M., AND REED, C. Automatic detection of arguments in legal texts. In *Proceedings of the 11th International Conference on Artificial Intelligence and Law* (New York, NY, USA, 2007), ICAIL '07, ACM, pp. 225–230.

[60] NATIONAL TELECOMMUNICATIONS AND INFORMATION ADMINISTRATION. Short form notice code of conduct to promote transparency in mobile app practices. `http://www.ntia.doc.gov/files/ntia/publications/july_25_code_draft.pdf`, July 2013. Last accessed: July 1, 2014.

[61] PASSONNEAU, R. Measuring Agreement on Set-valued Items (MASI) for semantic and pragmatic annotation. In *Proceedings of the international Conference on Language Resources and Evaluation* (2006), LREC '06.

[62] PASSONNEAU, R. J., AND CARPENTER, B. The benefits of a model of annotation. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse* (Stroudsburg, PA, USA, 2013), ACL, pp. 187–195.

[63] PORTER, M. An algorithm for suffix stripping. *Program: electronic library and information systems 14*, 3 (1980), 130–137.

[64] RAJARAMAN, A., AND ULLMAN, J. D. *Mining of massive datasets*. Cambridge University Press, New York, NY, USA, 2012.

[65] REEDER, R. W. *Expandable Grids: a user interface visualization technique and a policy semantics to support fast, accurate security and privacy policy authoring*. PhD thesis, Pittsburgh, PA, USA, 2008. AAI3321049.

[66] REEDER, R. W., KELLEY, P. G., MCDONALD, A. M., AND CRANOR, L. F. A user study of the Expandable Grid applied to P3P privacy policy visualization. In *Proceedings of the 7th ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2008), WPES '08, ACM, pp. 45–54.

[67] REIDENBERG, J. R. The use of technology to assure internet privacy: Adapting labels and filters for data protection. *Lex Electronica 3*, 2 (1997).

[68] REIDSMA, D., AND CARLETTA, J. Reliability measurement without limits. *Comput. Linguist. 34*, 3 (Sept. 2008), 319–326.

[69] RUBINSTEIN, I. S. Privacy and regulatory innovation: Moving beyond voluntary codes. *ISJLP 6*, 3 (2011), 355–423.

[70] STAMEY, J. W., AND ROSSI, R. A. Automatically identifying relations in privacy policies. In *27th ACM International Conference on Design of Communication* (New York, NY, USA, 2009), SIGDOC '09, ACM, pp. 233–238.

[71] STEDE, M., AND KUHN, F. Identifying the content zones of German court decisions. In *Lecture Notes in Business Information Processing* (Berlin, Heidelberg, Germany, 2009), vol. 37 of *BIS '09*, Springer, pp. 310–315.

[72] SYMANTEC CORPORATION. Complete privacy statement. `http://web.archive.org/web/20131028120625/http://www.symantec.com/about/profile/policies/privacy.jsp`. Last accessed: July 1, 2014.

[73] SYMANTEC CORPORATION. Global privacy statement for Symantec. `http://web.archive.org/web/19991012020231/http://symantec.com/legal/privacy.html`. Last accessed: July 1, 2014.

[74] TSOUMAKAS, G., AND KATAKIS, I. Multi label classification: An overview. *IJDWM 3*, 3 (2007), 1–13.

[75] WESTERHOUT, E. Definition extraction using linguistic and structural features. In *Proceedings of the 1st Workshop on Definition Extraction* (Stroudsburg, PA, USA, 2009), WDE '09, ACL, pp. 61–67.

[76] WESTERHOUT, E. Extraction of definitions using grammar-enhanced machine learning. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop* (Stroudsburg, PA, USA, 2009), EACL '09, ACL, pp. 88–96.

[77] WU, G., GREENE, D., AND CUNNINGHAM, P. Merging multiple criteria to identify suspicious reviews. In *Proceedings of the Fourth ACM Conference on Recommender Systems* (New York, NY, USA, 2010), RecSys '10, ACM, pp. 241–244.

[78] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A language for automatically enforcing privacy policies. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 85–96.

Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing

Matthew Fredrikson*, Eric Lantz*, Somesh Jha*, Simon Lin†, David Page*, Thomas Ristenpart*
University of Wisconsin, Marshfield Clinic Research Foundation*†

Abstract

We initiate the study of privacy in pharmacogenetics, wherein machine learning models are used to guide medical treatments based on a patient's genotype and background. Performing an in-depth case study on privacy in personalized warfarin dosing, we show that suggested models carry privacy risks, in particular because attackers can perform what we call *model inversion*: an attacker, given the model and some demographic information about a patient, can predict the patient's genetic markers.

As differential privacy (DP) is an oft-proposed solution for medical settings such as this, we evaluate its effectiveness for building private versions of pharmacogenetic models. We show that *DP mechanisms prevent our model inversion attacks when the privacy budget is carefully selected*. We go on to analyze the impact on utility by performing simulated clinical trials with DP dosing models. We find that for privacy budgets effective at preventing attacks, *patients would be exposed to increased risk of stroke, bleeding events, and mortality*. We conclude that *current* DP mechanisms do not simultaneously improve genomic privacy while retaining desirable clinical efficacy, highlighting the need for new mechanisms that should be evaluated *in situ* using the general methodology introduced by our work.

1 Introduction

In recent years, technical advances have enabled inexpensive, high-fidelity molecular analyses that characterize the genetic make-up of an individual. This has led to widespread interest in *personalized medicine*, which tailors treatments to each individual patient using genotype and other information to improve outcomes. Much of personalized medicine is based on *pharmacogenetic* (sometimes called *pharmacogenomic*) models [3, 14, 21, 40] that are constructed using supervised machine learning over large patient databases containing clinical and genomic data. Prior works [36, 37] in non-medical settings have shown that leaking datasets can enable de-anonymization of users and other privacy risks. In the pharmacogenetic setting, datasets themselves are often only disclosed to researchers, yet the models learned from them are made public (e.g., published in a paper). *Our focus is therefore on determining to what extent the models themselves leak private information, even in the absence of the original dataset.*

To do so, we perform a case study of warfarin dosing, a popular target for pharmacogenetic modeling. Warfarin is an anticoagulant widely used to help prevent strokes in patients suffering from atrial fibrillation (a type of irregular heart beat). However, it is known to exhibit a complex dose-response relationship affected by multiple genetic markers [43], with improper dosing leading to increased risk of stroke or uncontrolled bleeding [41]. As such, a long line of work [3, 14, 16, 21, 40] has sought pharmocogenetic models that can accurately predict proper dosage based on patient clinical history, demographics, and genotype. A review of this literature is given in [23].

Our study uses a dataset collected by the *International Warfarin Pharmocogenetics Consortium* (IWPC), to date the most expansive such database containing demographic information, genetic markers, and clinical histories for thousands of patients from around the world. While this particular dataset is publicly-available in a deidentified form, it is equivalent to data used in other studies that must be kept private (e.g., due to lack of consent to release). We therefore use it as a proxy for a private dataset. The paper authored by IWPC members [21] details methods to learn linear regression models from this dataset, and shows that using the resulting models to predict initial dose outperforms the standard clinical regimen in terms of absolute distance from stable dose. Randomized trials have been done to evaluate clinical effectiveness, but have not yet validated the utility of genetic information [27].

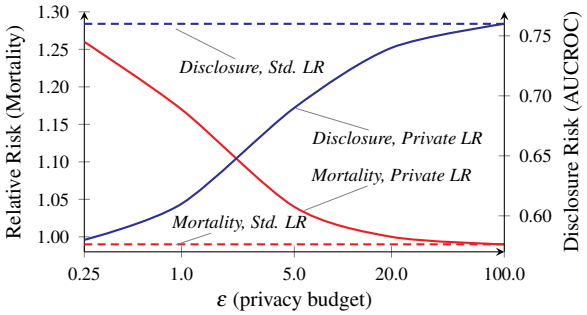


Figure 1: Mortality risk (relative to current clinical practice) for, and VKORC1 genotype disclosure risk of, ε-differentially private linear regression (LR) used for warfarin dosing (over five values of ε, curves are interpolated). Dashed lines correspond to non-private linear regression.

Model inversion. We study the degree to which these models leak sensitive information about patient genotype, which would pose a danger to genomic privacy. To do so, we investigate *model inversion attacks* in which an adversary, given a model trained to predict a specific variable, uses it to make predictions of unintended (sensitive) attributes used as input to the model (i.e., an attack on the privacy of attributes). Such attacks seek to take advantage of correlation between the target, unknown attributes (in our case, demographic information) and the model output (warfarin dosage). A priori it is unclear whether a model contains enough exploitable information about these correlations to mount an inversion attack, and it is easy to come up with examples of models for which attackers will not succeed.

We show, however, that warfarin models do pose a privacy risk (Section 3). To do so, we provide a general model inversion algorithm that is optimal in the sense that it minimizes the attacker's *expected misprediction rate* given the available information. We find that when one knows a target patient's background and stable dosage, their genetic markers are predicted with significantly better accuracy (up to 22% better) than guessing based on marginal distributions. In fact, *it does almost as well as regression models specifically trained to predict these markers (only ˜5% worse)*, suggesting that model inversion can be nearly as effective as learning in an "ideal" setting. Lastly, the inverted model performs measurably better for members of the training cohort than others (yielding an increased 4% accuracy) indicating a leak of information specifically about those patients.

Role of differential privacy. Differential privacy (DP) is a popular framework for designing statistical release mechanisms, and is often proposed as a solution to privacy concerns in medical settings [10, 12, 45, 47]. DP is parameterized by a value ε (sometimes referred to as the *privacy budget*), and a DP mechanism guarantees that the likelihood of producing any particular output from an input cannot vary by more than a factor of e^{ε} for "similar" inputs differing in only one subject.

Following this definition in our setting, DP guarantees protection against attempts to infer whether a subject was included in the training set used to derive a machine learning model. It does *not* explicitly aim to protect attribute privacy, which is the target of our model inversion attacks. However, others have motivated or designed DP mechanisms with the goal of ensuring the privacy of patients' diseases [15], features on users' social network profiles [33], and website visits in network traces [38]—all of which relate to attribute privacy. Furthermore, recent theoretical work [24] has shown that in some settings, including certain applications of linear regression, incorporating noise into query results preserves attribute privacy. This led us to ask: can *genomic* privacy benefit from the application of DP mechanisms in our setting?

To answer this question, we performed the first end-to-end evaluation of DP in a medical application (Section 5). We employ two recent algorithms on the IWPC dataset: the *functional mechanism* of Zhang *et al.* [47] for producing private linear regression models, and Vinterbo's *privacy-preserving projected histograms* [44] for producing differentially-private synthetic datasets, over which regression models can be trained. These algorithms represent the current state-of-the-art in DP mechanisms for their respective models, with performance reported by the authors that exceeds previous DP mechanisms designed for similar tasks.

On one end of our evaluation, we apply a model inverter to quantify the amount of information leaked about patient genetic markers by ε-DP versions of the IWPC model. On the other end, we quantify the impact of ε on patient outcomes, performing *simulated clinical trials* via techniques widely used in the medical literature [4, 14, 18, 19]. Our main results, a subset of which are shown in Figure 1, show a clear trade-off between patient outcomes and privacy:

- ***"Small ε"-DP protects genomic privacy***: Even though DP was not specifically designed to protect attribute privacy, we found that for sufficiently small ε (≤ 1), genetic markers cannot be accurately predicted (see the line labeled "Disclosure, private LR" in Figure 1), and there is no discernible difference between the model inverter's performance on the training and validation sets. However, this effect quickly vanishes as ε increases, where genotype is predicted with up to 58% accuracy (0.76 AUCROC). This is significantly (22%) better than the 36% accuracy one achieves without the models, and not far below (5%) the "best possible" performance of a non-private regression model trained to predict the same genotype using IWPC data.

- ***Current DP mechanisms harm clinical efficacy***: Our simulated clinical trials reveal that for $\varepsilon \leq 5$ the risk of fatalities or other negative outcomes increases significantly (up to $1.26\times$) compared to the current clinical practice, which uses non-personalized, fixed dosing and so leaks no information at all. Note that the range of ε (> 5) that provides clinical utility not only fails to protect genomic privacy, but are commonly assumed to provide insufficient DP guarantees as well. (See the line labeled "Mortality, private LR" in Figure 1.)

Put simply: our analysis indicates that in this setting where utility is paramount, the best known mechanisms for our application do not give an ε for which state-of-the-art DP mechanisms can be reasonably employed.

Implications of our results. Our results suggest that there is still much to learn about pharmacogenetic privacy. Differential privacy is suited to settings in which *privacy and utility requirements are not fundamentally at odds*, and can be balanced with an appropriate privacy budget. Although the mechanisms we studied do not properly strike this balance, future mechanisms may be able to do so—the *in situ* methodology given in this paper may help to guide such efforts. In settings where privacy and utility *are* fundamentally at odds, release mechanisms of any kind will fail, and restrictive access control policies may be the best answer. The model inversion techniques outlined here can help to identify these situations, and quantify the risks.

2 Background

Warfarin and Pharmacogenetics Warfarin, also known in the United States by the brand name Coumadin, is a widely prescribed anticoagulant medication. It is used to treat patients suffering from cardiovasvular problems, including atrial fibrillation (a type of irregular heart beat) and heart valve replacement. By reducing the tendency of blood to clot, at appropriate dosages it can reduce risk of clotting events, particularly stroke. Unfortunately, warfarin is also very difficult to dose: proper dosages can differ by an order of magnitude between patients, and this has led to warfarin's status as one of the leading causes of drug-related adverse events in the United States [26]. Underestimating the dose can result in failure to prevent the condition the drug was prescribed to treat. Overestimating the dose can, just as seriously, lead to uncontrolled bleeding events because the drug interferes with clotting. Because of these risks, in existing clinical practice patients starting on warfarin are given a fixed initial dose but then must visit a clinic many times over the first few weeks or months of treatment in order to determine the correct dosage which gives the desired therapeutic effect.

Stable dose is assessed clinically by measuring the time it takes for blood to clot, called prothrombin time. This measure is standardized between different manufacturers as an international normalized ratio (INR). Based on the patient's indication for (i.e., the reason to prescribe) warfarin, a clinician determines a target INR range. After the fixed initial dose, later doses are modified until the patient's INR is within the desired range and maintained at that level. INR in the absence of anticoagulation therapy is approximately 1, while the desired INR for most patients in anticoagulation therapy is in the range 2–3 [5]. INR is the response measured by the physiological model used in our simulations in Section 5.

Genetic variability among patients is known to play an important role in determining the proper dose of warfarin [23]. Polymorphisms in two genes, VKORC1 and CYP2C9, are associated with the mechanism with which the body metabolizes the drug, which in turn affects the dose required to reach a given concentration in the blood. Warfarin works by interfering with the body's ability to recycle vitamin K, which is used to regulate blood coagulation. VKORC1, part of the vitamin K epoxide reductase complex, is a component of the vitamin K cycle. CYP2C9 encodes for a variant of cytochrome P450, a family of proteins which oxidize a variety of medications. Since each person has two copies of each gene, there are several combinations of variants possible. Following the IWPC paper [21], we represent VKORC1 polymorphisms by single nucleotide polymorphism (SNP) rs9923231, which is either G (common variant) or A (uncommon variant), resulting in three combinations G/G, A/G, or A/A. Similarly, CYP2C9 variants are *1 (most common), *2, or *3, resulting in 6 combinations.

Taken together with age and height, Sconce *et al.* [40] demonstrated that CYP2C9 and VKORC1 account for 54% of the total warfarin dose requirement variability. In turn, a large literature (over 50 papers as of early 2013) has sought pharmacogenetic algorithms that predict proper dose by taking advantage of patient genetic markers for CYP2C9 and VKORC1, together with demographic information and clinical history (e.g., current medications). These typically involve learning a simple predictive model of stable dose from previously obtained outcomes. We focus on the IWPC algorithm [21], a study resulting in production of a linear regression model that, when used to predict the initial dosage, has been shown to provide improved outcomes in simulated clinical trials using the IWPC dataset discussed below. Interestingly, linear regression performed as well or better than a wide variety of other, more complex machine learning techniques. Some pharmacogenetic algorithms for warfarin are currently also undergoing (real) clinical trials [1].

Dataset The IWPC [21] collected data on patients who were prescribed warfarin from 21 sites in 9 countries on 4 continents. The data was curated by staff at the Pharmacogenomics Knowledge Base [2], and each site obtained informed consent to use de-identified data from patients prior to the study. Because the dataset contains no protected health information, and the Pharmacogenomics Knowledge Base has since made the dataset publicly available for research purposes, it is exempt from institutional review board review. However, the type of data contained in the IWPC dataset is equivalent to many other medical datasets that have not been released publicly [3, 7, 16, 40], and are considered private.

Each patient was genotyped for at least one SNP in VKORC1, and for variants of CYP2C9. In addition, other information such as age, height, weight, race, and other medications was collected. The outcome variable is the stable therapeutic dose of warfarin, defined as the steady-state dose that led to stable anticoagulation levels. The patients in our dataset were restricted to those with target INR in the range 2–3 (the vast majority of patients), as is standard practice with most studies of warfarin dosing efficacy [3, 14]. We divided the data into two cohorts based on those used in IWPC [21]. The first (training) cohort was used to build a set of pharmacogenetic dosing algorithms. The second (validation) cohort was used to test privacy attacks as well as draw samples for the clinical simulations. To make the data suitable for regression we removed all patients missing CYP2C9 or VKORC1 genotype, normalized the data to the range [-1,1], converted all nominal attributes into binary-valued numeric attributes, and scaled each row into the unit sphere. Our eventual training cohort consisted of 2644 patients, and our validation cohort of 853 patients, and corresponds to the same training-validation split used by IWPC (but without the missing values used in the IWPC split).

3 Privacy of Pharmacogenetic Models

In this section we investigate the risks involved in releasing regression models trained over private data, using models that predict warfarin dose as our case study. We consider a setting where an adversary is given access to such a model, the warfarin dosage of an individual, some rudimentary information about the data set, and possibly some additional attributes about that individual. The adversary's goal is to predict one of the *genotype* attributes for that individual. In order for this setting to make sense, the genotype attributes, warfarin dose, and other attributes known to the adversary must all have been in the private data set. We emphasize that the techniques introduced can be applied more generally, and save as future work investigating other pharmacogenetic settings.

3.1 Attack Model

We assume an adversary who employs an inference algorithm $\mathcal{A}$ to discover the genotype (in our experiments, either CYP2C9 or VKORC1) of a target individual α. The adversary has access to a linear model f trained over a dataset D drawn i.i.d. from an unknown prior distribution p. D has domain $\mathbf{X} \times Y$, where $\mathbf{X} = X_1, \ldots, X_d$ is the domain of possible *attributes* and Y is the domain of the *response*. α is represented by a single row in D, $(\mathbf{x}^\alpha, y^\alpha)$, and the attribute learned by the adversary is referred to as the *target attribute* $\mathbf{x}_t^\alpha$.

In addition to f, the adversary has access to marginals[1] $p_{1,\ldots,d,y}$ of the joint prior p, the dataset domain $\mathbf{X} \times Y$, α's stable dosage y^α of warfrain, some information π about f's performance (details in the following section), and either of the following subsets $\mathbf{x}_K^\alpha$ of α's attributes:

- *Basic demographics*: a subset of α's demographic data, including age (binned into eight groups by the IWPC), race, height, and weight (denoted $x_{\text{age}}^\alpha, x_{\text{race}}^\alpha, \ldots$). Note that this corresponds to a subset of the non-genetic attributes in D.
- *All background*: all of p's attributes except CYP2C9 or VKORC1 genotype.

The adversary has black-box access to f. Unless it is clear from the context, we will specify whether f is the output of a DP mechanism, and which type of background information is available.

3.2 Model Inversion

In this section, we discuss a technique for inferring CYP2C9 and VKORC1 genotype from a model designed to predict warfarin dosing. Given a model f that takes inputs $\mathbf{x}$ and outputs a predicted stable dose y, the attacker seeks to build an algorithm $\mathcal{A}$ that takes as input some subset $\mathbf{x}_K^\alpha$ of attributes (corresponding to demographic or additional background attributes from $\mathbf{X}$), a known stable dose y^α, and outputs a prediction of $\mathbf{x}_t$ (corresponding either to CYP2C9 or VKORC1). We begin by presenting a general-purpose algorithm, and show how it can be applied to linear regression models.

A general algorithm. We present an algorithm for model inversion that is independent of the underlying model structure (Figure 2). The algorithm works by estimating the probability of a potential target attribute given the available information and the model. Its operation is straightforward: *candidate* database rows that are similar to what is known about α are run *forward* through

[1] These are commonly published in studies, and when it is clear from the context, we will drop the subscript.

1. Input: $\mathbf{z}_K = (x_1, \ldots, x_k, y), f, p_{1,\ldots,d,y}$
2. Find the *feasible set* $\hat{\mathbf{X}} \subseteq \mathbf{X}$, i.e., such that $\forall \mathbf{x} \in \hat{\mathbf{X}}$
   (a) $\mathbf{x}$ matches $\mathbf{z}_K$ on known attributes: for $1 \le i \le k, \mathbf{x}_i = x_i$.
   (b) f evaluates to y as given in $\mathbf{z}_K$: $f(\mathbf{x}) = y$.
3. If $|\hat{\mathbf{X}}| = 0$, return $\perp$.
4. Return x_t that maximizes $\sum_{\mathbf{x} \in \hat{\mathbf{X}}: \mathbf{x}_t = x_t} \prod_{1 \le i \le d} p_i(\mathbf{x}_i)$

(a) $\mathcal{A}_0$: Model inversion without performance statistics.

1. Input: $\mathbf{z}_K = (x_1, \ldots, x_k, y)$, f, π, $p_{1,\ldots,d,y}$
2. Find the *feasible set* $\hat{\mathbf{X}} \subseteq \mathbf{X}$, i.e., such that $\forall \mathbf{x} \in \hat{\mathbf{X}}$
   (a) $\mathbf{x}$ matches $\mathbf{z}_K$ on known attributes: for $1 \le i \le k$, $\mathbf{x}_i = x_i$.
3. If $|\hat{\mathbf{X}}| = 0$, return $\perp$.
4. Return x_t that maximizes $\sum_{\mathbf{x} \in \hat{\mathbf{X}}: \mathbf{x}_t = x_t} \pi_{y, f(\mathbf{x})} \prod_{1 \le i \le d} p_i(\mathbf{x}_i)$

(b) $\mathcal{A}_\pi$: Model inversion with performance statistics π.

Figure 2: Model inversion algorithm.

the model. Based on the known priors, and how well the model's output on that row coincides with α's known response value, the candidate rows are weighted. The target attribute with the greatest weight, computed by marginalizing the other attributes, is returned.

Below, we describe this algorithm in more detail. We derive each step by showing how to compute the *least biased* estimate of the target attribute's likelihood, which the model inversion algorithm maximizes to form a prediction. As we reason below, this approach is *optimal* in the sense that it minimizes the expected misclassification rate when the adversary has no other information (i.e., makes no further assumptions) beyond what is given in Section 3.1.

Derivation. We begin the description with a simpler restricted case in which the model always produces the correct response. Assume for now that f is *perfect*, i.e., it never makes a misprediction, and we can assume that $f(\mathbf{x}) = y$ almost surely for any sample $(\mathbf{x}, y)$; this case is covered by $\mathcal{A}_0$ in Figure 2. In the following, we assume the sample corresponds to the individual α, and drop the superscript for clarity. Suppose the adversary wishes to learn the probability that $\mathbf{x}_t$ takes a certain value x_t, i.e., $\mathbf{Pr}[\mathbf{x}_t = x_t | \mathbf{x}_K, y]$, given some known attributes $\mathbf{x}_K$, response variable y, and the model f. Here, and in the following discussion, the probabilities in $\mathbf{Pr}[\cdot]$ expressions are always over draws from the unknown joint prior p unless stated otherwise. Let $\hat{\mathbf{X}} = \{\mathbf{x}' : \mathbf{x}'_K = \mathbf{x}_K \text{ and } f(\mathbf{x}') = y\}$ be the subset of $\mathbf{X}$ matching the given information $\mathbf{x}_K$ and y. Then by straightforward computation,

$$\mathbf{Pr}[x_t | \mathbf{x}_K, y] = \frac{\mathbf{Pr}[x_t, \mathbf{x}_K, y]}{\mathbf{Pr}[\mathbf{x}_K, y]} = \frac{\sum_{\mathbf{x}' \in \hat{\mathbf{X}}: \mathbf{x}'_t = x_t} p(\mathbf{x}', y)}{\sum_{\mathbf{x}' \in \hat{\mathbf{X}}} p(\mathbf{x}', y)} \quad (1)$$

Now, the adversary does not know the true underlying joint prior p. He only knows the marginals $p_{1,\ldots,d,y}$, so any distribution with these marginals is a possible prior. To characterize the unbiased prior that satisfies these constraints, we apply the *prinicipal of maximum entropy*[2] [22], which in our setting gives the prior:

$$p(\mathbf{x}, y) = p(y) \cdot \prod_{1 \le i \le d} p(\mathbf{x}_i) \quad (2)$$

Continuing with the previous expression, we now have,

$$\mathbf{Pr}[x_t | \mathbf{x}_K, y] = \frac{\sum_{\mathbf{x}' \in \hat{\mathbf{X}}: \mathbf{x}'_t = x_t} p(y) \prod_i p(\mathbf{x}'_i)}{\sum_{\mathbf{x}' \in \hat{\mathbf{X}}} p(y) \prod_i p(\mathbf{x}'_i)} \quad (3)$$

$$\propto \sum_{\mathbf{x}' \in \hat{\mathbf{X}}: \mathbf{x}'_t = x_t} \prod_i p(\mathbf{x}'_i) \quad (4)$$

This last step follows because the denominator is independent of the choice of x_t. Notice that this is exactly the quantity that is maximized by the value returned by $\mathcal{A}_0$ (Figure 2 (a)). This is the *maximum a posteriori probability* (MAP) estimate, which minimizes the adversary's expected misclassification rate. Under these assumptions, $\mathcal{A}_0$ is an optimal algorithm for model inversion.

$\mathcal{A}_\pi$ in Figure 2 (b) generalizes this reasoning to the case where f is not assumed to be perfect, and the adversary has information about the performance of f over samples drawn from p. We model this information with a function π, defined in terms of a random sample $\mathbf{z}$ from p,

$$\pi(y, y') = \mathbf{Pr}\left[\mathbf{z}_y = y | f(\mathbf{z}_\mathbf{x}) = y'\right] \quad (5)$$

In other words, $\pi(y, y')$ gives the probability that the true response drawn with attributes $\mathbf{z}_\mathbf{x}$ is y given that the model outputs y'. We write $\pi_{y,y'}$ to simplify notation. In practice, π can be estimated using statistics commonly released with models, such as confusion matrices or standardized regression error.

Because f is not assumed to be perfect in the general setting, $\hat{\mathbf{X}}$ is defined slightly differently than in $\mathcal{A}_0$; the second restriction, that $f(\mathbf{x}^\alpha) = y^\alpha$, is removed. After constructing $\hat{\mathbf{X}}$, $\mathcal{A}_\pi$ uses the marginals and π to weight each candidate $\mathbf{x} \in \hat{\mathbf{X}}$ by the probability that f behaves as observed (i.e., outputs $f(\mathbf{x})$) when the response variable matches what the adversary knows to be true (i.e.,

[2] *cf.* Jaynes [22], "[The maximum entropy prior] is least biased estimate possible on the given information; i.e., it is maximally noncommittal with regard to missing information."

y). Again, using the maximum entropy prior from before gives the MAP estimate in the more general setting,

$$\mathbf{Pr}[x_t|\mathbf{x}_K, y^\alpha, f] = \frac{\sum_{\mathbf{x}'\in\hat{\mathbf{X}}:\mathbf{x}'_t=x_t} \mathbf{Pr}[\mathbf{x}', y, f(\mathbf{x}')]}{\sum_{\mathbf{x}'\in\hat{\mathbf{X}}} \mathbf{Pr}[\mathbf{x}', y, f(\mathbf{x}')]} \quad (6)$$

$$= \frac{\sum_{\mathbf{x}'\in\hat{\mathbf{X}}:\mathbf{x}'_t=x_t} \mathbf{Pr}[y|\mathbf{x}', f(\mathbf{x}')]\, p(\mathbf{x}')}{\sum_{\mathbf{x}'\in\hat{\mathbf{X}}} \mathbf{Pr}[\mathbf{x}', y, f(\mathbf{x}')]} \quad (7)$$

$$\propto \sum_{\mathbf{x}'\in\hat{\mathbf{X}}:\mathbf{x}'_t=x_t} \pi_{y,f(\mathbf{x}')} \left(\prod_i p(\mathbf{x}'_i)\right) \quad (8)$$

The second step follows from the independence of the maximum entropy prior in our setting, and the fact that $\mathbf{x}$ determines $f(\mathbf{x})$ so $\mathbf{Pr}[f(\mathbf{x}'), \mathbf{x}'] = \mathbf{Pr}[\mathbf{x}']$.

Application to linear regression. Recall that a linear regression model assumes that the response is a linear function of the attributes, i.e., there exists a coefficient vector $\mathbf{w} \in \mathbb{R}^d$ and random *residual error* δ such that $y = \mathbf{w}^T\mathbf{x} + b + \delta$ for some bias term b. A linear regression model f_L is then an estimate $(\hat{\mathbf{w}}, \hat{b})$ of $\mathbf{w}$ and the bias term, which operates as: $f_L(\mathbf{x}) = \hat{b} + \hat{\mathbf{w}}^T\mathbf{x}$. It is typical to assume that δ has a fixed Gaussian distribution $\mathcal{N}(0, \sigma^2)$ for some variance σ. Most regression software estimates σ^2 empirically from training data, so it is often published alongside a linear regression model. Using this the adversary can derive an estimate of π,

$$\hat{\pi}(y, y') = \mathbf{Pr}_{\mathcal{N}(0,\sigma^2)}[y - y']$$

Steps 2 and 4 of $\mathcal{A}_\pi$ may be expensive to compute if $|\hat{\mathbf{X}}|$ is large. In this case, one can approximate using Monte Carlo techniques to sample members of $\hat{\mathbf{X}}$. Fortunately, in our setting, the nominal-valued variables all come from sets with small cardinality. The continuous variables have natural discretizations, as they correspond to attributes such as age and weight. Thus, step 4 can be computed directly by taking a discrete convolution over the unknown attributes without resorting to approximation.

Discussion. We have argued that $\mathcal{A}_\pi$ is optimal in one particular sense, i.e., it minimizes the expected misclassification rate on the maximum-entropy prior given the available information (the model and marginals). However, it is not hard to specify joint priors p for which the marginals $p_{1,\ldots,d,y}$ convey little useful information, so the expected misclassification rate minimized here diverges substantially from the true rate. In these cases, $\mathcal{A}_\pi$ may perform poorly, and more background information is needed to accurately predict model inputs.

There is also the possibility that the model itself does not contain enough useful information about the correlation between certain input attributes and the output. For illustrative purposes, consider a model taking one input attribute, that discards all information about that attribute except a single bit, e.g., it performs a comparison with a fixed constant. If the attribute is distributed uniformly across a large domain, then $\mathcal{A}_\pi$ will only perform negligibly better than guessing from the marginal. Thus, determining how well a model allows one to predict sensitive inputs generally requires further analysis, which is the purpose of the evaluation that we discuss next (see also Section 4).

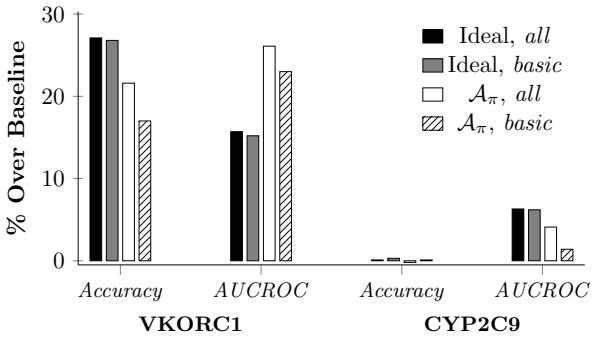


Figure 3: Model inversion performance, as improvement over baseline guessing from marginals, given a linear model derived from the training data. Available background information specified by *all* and *basic* as discussed in Section 3.1.

Results on non-private regression. To evaluate $\mathcal{A}_\pi$, we split the IWPC dataset into a training and validation set (see Section 2), D_T and D_V respectively, use D_T to derive a least-squares linear model f, and then run $\mathcal{A}_\pi$ on every α in D_T with either of the two background information types (*all* or *basic*, see Section 3.1) to predict both genotypes. In order to determine how how well one can predict these genotypes in an *ideal* setting, we built and evaluated a multinomial logistic regression model (using R's `nnet` package) for each genotype from the IWPC data. This allows us to compare the performance of $\mathcal{A}_\pi$ against "best-possible" results achieved using standard machine learning techniques with linear models.

We measure performance both in terms of *accuracy*, which is the percentage of samples for which the algorithm correctly predicted genotype, and *AUCROC*, which is the multi-class area under the ROC curve defined by Hand and Till [17]. While accuracy is generally easier to interpret, it can give a misleading characterization of predictive ability for *skewed* distributions—if the predicted attribute takes a particular value in 75% of the samples, then a trivial algorithm can easily obtain 75% accuracy by always guessing this value. AUCROC does not suffer this limitation, and so gives a more balanced characterization of how well an algorithm predicts both common and rare values.

The results are given in Figure 3, which shows the performance of $\mathcal{A}_\pi$ and "ideal" multinomial regression predicting VKORC1 and CYP2C9 on the training set. The numbers are given relative to the baseline performance obtained by always guessing the most probable genotype based on the given marginal prior–36% accuracy on VKORC1, 75% accuracy on CYP2C9, and 0.5 AUCROC for both genotypes. We see that $\mathcal{A}_\pi$ comes close to ideal accuracy on VKORC1 (5% less accurate with all background information), and actually exceeds the ideal predictor in terms of AUCROC. This means that $\mathcal{A}_\pi$ does a better job predicting *rare* genotypes than the ideal model, but does slightly worse overall, and may be a result of the ideal model avoiding overfitting to uncommon data points.

The results for CYP2C9 are quite different. Neither $\mathcal{A}_\pi$ or the ideal model were able to predict this genotype more accurately than baseline. This indicates that CYP2C9 is difficult to predict using linear models, and because we use a linear model to run $\mathcal{A}_\pi$ in this case, it is no surprise that it inherits this limitation. Both the ideal model and $\mathcal{A}_\pi$ slightly outperform baseline prediction in terms of AUCROC, and $\mathcal{A}_\pi$ comes very close to ideal performance (within 2%). In one case $\mathcal{A}_\pi$ does slightly worse (0.2%) than baseline accuracy; this may be due to the fact that the marginals and $\hat{\pi}$ used by $\mathcal{A}_\pi$ are approximations to the true marginals and error distribution π.

We also evaluated $\mathcal{A}_\pi$ on the validation set (using a model f derived from the training set). We found that both genotypes are predicted more accurately on the training set than validation. For VKORC1, $\mathcal{A}_\pi$ was 3% more accurate and yielded an additional 4% AUCROC. The difference was less pronounced with CYP2C9, which was 1.5% more accurate with an additional 2% AUCROC. Although these differences are not as large as the aboslute gain over baseline prediction, they persist across other training/validation splits. We ran 100 instances of cross-validation, and measured the difference between training and validation performance. We found that we were on average able to better predict the training cohort ($p < 0.01$).

4 Differentially-Private Mechanisms and Pharmacogenetics

In the last section, we saw that linear models trained on private datasets leak information about patients in the training cohort. In this section, we explore the issue on models and datasets for which differential privacy has been applied.

As in the previous section, we take the perspective of the adversary, and attempt to infer patients' genotype given differentially-private models and different types of background information on the targeted individual. As such, we use the same attack model, but rather than assuming the adversary has access to f, we assume access to a *differentially private version of* the original dataset D or f. We use two published differentially-private mechanisms with publicly-available implementations: *private projected histograms* [44] and the *functional mechanism* [47] for learning private linear regression models. Although full histograms are typically not published in pharmacogenetic studies, we analyze their privacy properties here to better understand the behavior of differential privacy across algorithms that implement it differently.

Our key findings are summarized as follows:

- Some ε values effectively protect genomic privacy for DP linear regression. For $\varepsilon \leq 1$, $\mathcal{A}_\pi$ could not predict VKORC1 better on the training set than the validation set either in terms of accuracy or AUCROC. The same result holds on CYP2C9, but only when measured in terms of AUCROC. $\mathcal{A}_\pi$'s *absolute* performance for these ε is not much better than the baseline either: VKORC1 is predicted only 5% better at $\varepsilon = 1$, and CYP2C9 sees almost no improvement.

- "Large"-ε DP mechanisms offer little genomic privacy. When $\varepsilon \geq 5$, both DP mechanisms see a statistically-significant increase in training set performance over validation ($p < 0.02$), and as ε approaches 20 there is little difference from non-private mechanisms (between 3%-5%).

- Private histograms disclose significantly more information about genotype than private linear regression, even at identical ε values. At all tested ε, private histograms leaked more on the training than validation set. *This result holds even for non-private regression models*, where the AUCROC gap reached 3.7% area under the curve, versus the 3.9% - 5.9% gap for private histograms. This demonstrates that the relative nature of differential privacy's guarantee can lead to meaningful concerns.

Our results indicate that understanding the implications of differential privacy for pharmacogenomic dosing is a difficult matter—even small values of ε might lead to unwanted disclosure in many cases.

Differential Privacy Dwork introduced the notion of differential privacy [11] as a constructive response to an impossibility result concerning stronger notions of private data release. For our purposes, a dataset D is a number m of vector, value pairs $(\mathbf{x}^{\alpha_1}, y^{\alpha_1}), \ldots, (\mathbf{x}^{\alpha_m}, y^{\alpha_m})$

where $\alpha_1, \ldots, \alpha_m$ are (randomized) patient identifiers, each $\mathbf{x}^{\alpha_i} = [x_1^{\alpha_i}, \ldots, x_d^{\alpha_i}]$ is a patient's demographic information, age, genetic variants, etc., and y^{α_i} is the stable dose for patient α_i. A (differential) privacy mechanism K is a randomized algorithm that takes as input a dataset D and, in the cases we consider, either outputs a new dataset D_{priv} or a linear model M_{priv} (i.e., a real- valued linear function with n inputs. We denote the set of possible outputs of a mechanism as $\text{Range}(\mathsf{K})$.

A mechanism K achieves ε-differential privacy if for all databases D_1, D_2 differing in at most one row, and all $S \subseteq \text{Range}(\mathsf{K})$,

$$\Pr[\mathsf{K}(D_1) \in S] \leq \exp(\varepsilon) \times \Pr[\mathsf{K}(D_2) \in S]$$

Differential privacy is an information-theoretic guarantee, and holds regardless of the auxiliary information an adversary posesses about the database.

Differentially-private histograms. We first investigate a mechanism for creating a differentially-private version of a dataset via the private projected histogram method [44]. DP datasets are appealing because an (untrusted) analyst can operate with more freedom when building a model; he is free to select whichever algorithm or representation best suits his task, and need not worry about finding differentially-private versions of the best algorithms.

Because the numeric attributes in our dataset are too fine-grained for effective histogram computation, we first discretize each numeric attribute into equal-width bins. In order to select the number of bins, we use a heuristic given by Lei [32] and suggested by Vinterbo [44], which says that when numeric attributes are scaled to the interval $[0,1]$, the bin width is given by $(\log(n)/n)^{1/(d+1)}$, where $n = |D|$ and d is the dimension of D. In our case, this is implies two bins for each numeric attribute. We validated this parameter against our dataset by constructing 100 differentially-private datasets at $\varepsilon = 1$ with $2, 3, 4$, and 5 bins for each numeric attribute, and measured the accuracy of a dose-predicting linear regression model over each dataset. The best accuracy was given for $k = 2$, with the difference in means for $k = 2$ and $k = 3$ not attributable to noise. When the discretized attributes are translated into a private version of the original dataset, the median value from each bin is used to create numeric values.

To infer the private genomic attributes given a differentially-private version D_ε of a dataset, we can compute an empirical approximation $\hat{p}$ to the joint probability distribution p (see Section 3.1) by counting the frequency of tuples in D_ε. A minor complication arises due to the fact that numeric values in D_ε have been discretized and re-generated from the median of each bin. Therefore, the likelihood of finding a row in D_ε that matches any row in D_T or D_V is low. To account for this, we transform each numeric attribute in the background information to the nearest median from the corresponding attribute used in the discretization step when generating D_ε. We then use $\hat{p}$ to directly compute a prediction of the genotype x_t that maximizes $\Pr_{\hat{p}}[\mathbf{x}_t^\alpha = x_t | \mathbf{x}_K^\alpha, y^\alpha]$.

Differentially-private linear regression. We also investigate use of the functional mechanism [47] for producing differentially-private linear regression models. The functional mechanism works by adding Laplacian noise to the coefficients of the objective function used to drive linear regression. This technique stands in contrast to the more obvious approach of directly perturbing the output coefficients of the regression training algorithm, which would require an explicit sensitivity analysis of the training algorithm itself. Instead, deriving a bound on the amount of noise needed for the functional mechanism involves a fairly simple calculation on the objective function [47].

We produce private regression models on the IWPC dataset by first projecting the columns of the dataset into the interval $[-1, 1]$, and then scaling the non-response columns (i.e., all those except the patient's dose) of each row into the unit sphere. This is described in the paper [47] and performed in the publicly-available implementation of the technique, and is necessary to ensure that sufficient noise is added to the objective function (i.e., the amount of noise needed is not scale-invariant). In order to inter-operate with the other components of our evaluation apparatus, we re-implemented the algorithm in R by direct translation from the authors' Matlab implementation. We evaluated the accuracy of our implementation against theirs, and found no statistically-significant difference.

Applying model inversion to the functional mechanism is straightforward, as our technique from Section 3.2 makes no assumptions about the internal structure of the regression model or how it was derived. However, care must be taken with regards to data scaling, as the functional mechanism classifier is trained on scaled data. When calculating $\hat{X}$, all input variables must be transformed by the same scaling function used on the training data, and the predicted response must be transformed by the inverse of this function.

Results on private models. We evaluated our inference algorithms on both mechanisms discussed above at a range of ε values: 0.25, 1, 5, 20, and 100. For each algorithm and ε, we generated 100 private models on the training cohort, and attempted to infer VKORC1 and CYP2C9 for each individual in both the training and validation cohort. All computations were performed in R.

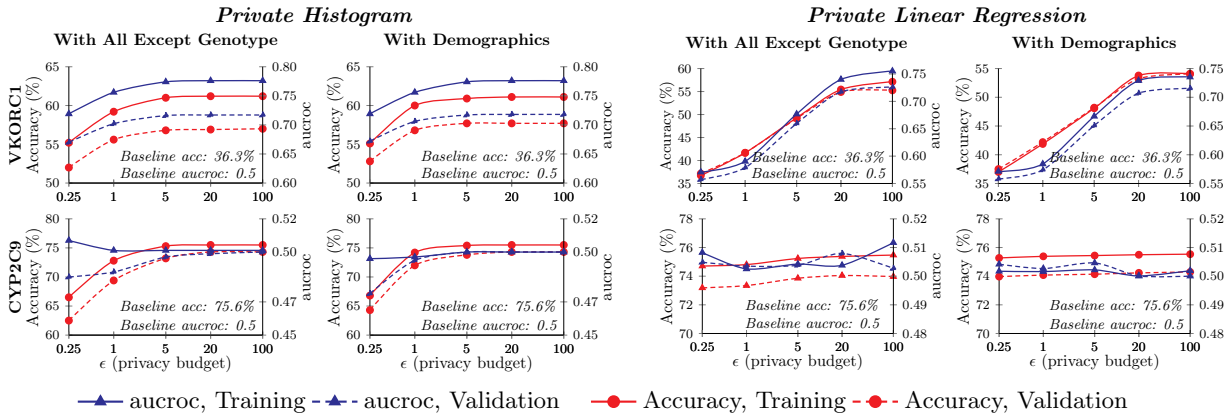


Figure 4: Inference performance for genomic attributes over IWPC training and validation set for private histograms (**left**) and private linear regression (**right**), assuming both configurations for background information. Dashed lines represent accuracy, solid lines area under the ROC curve (AUCROC).

Figure 4 shows our results in detail. In the following, we discuss the main takeaway points.

Private Histograms vs. Linear Regression. We found that private histograms leaked significantly more information about patient genotype than private linear regression models. The difference in AUCROC for histograms versus regression models is statistically significant for VKORC1 at all ε. As Figure 4 indicates, the magnitude of the difference from baseline is also higher for histograms when considering VKORC1, nearly reaching 0.8 AUCROC and 63% accuracy, while regression models achieved at most 0.75 AUCROC and 55–60% accuracy. The AUCROC performance for VKORC1 was greater than the baseline for all ε. However, for CYP2C9 this result only held when assuming all background information except genotype, and only for $\varepsilon \leq 5$; when we assumed only demographic information, there was no significant difference between baseline and private histogram performance.

Disclosure from Overfitting. In nearly all cases, we were able to better infer genotype for patients in the training set than those in the validation set. For private linear regression, this result holds for VKORC1 at $\varepsilon \geq 5.0$ for AUCROC. This is not an artifact of the training/validation split chosen by the IWPC; we ran 10-fold cross validation 100 times, measuring the AUCROC difference between training and test set validation, and found a similar difference between training and validation set performance ($p < 0.01$). The fact that the difference at certain ε values is not statistically significant is evidence that private linear regression is effective at preventing genotype disclosure at these ε. For private histograms, this result held for VKORC1 at all ε, and CYP2C9 at $\varepsilon < 5$ with all background information but genotype.

Differences in Genotype. For both private regression and histogram models, performance for CYP2C9 is strikingly lower than for VKORC1. Private regression models performed no differently from the baseline, achieving essentialy no gain in terms of accuracy and at most 1% gain in AUCROC. We observe that this also held in the non-private setting, and the ideal model achieved the same accuracy as baseline, and only 7% greater AUCROC. This indicates that CYP2C9 is not well-predicted using linear models, and $\mathcal{A}_\pi$ performed nearly as well as is possible.

5 The Cost of Privacy: Negative Outcomes

In addition to privacy, we are also concerned with the utility of a warfarin dosing model. The typical approach to measuring this is a simple accuracy comparison against known stable doses, but ultimately we're interested in how errors in the model will affect patient health. In this section, we evaluate the potential medical consequences of using a differentially-private regression algorithm to make dosing decisions in warfarin therapy. Specifically, we estimate the increased risk of stroke, bleeding, and fatality resulting from the use of differentially-private warfarin dosing at several privacy budget settings. This approach differs from the normal methodology used for evaluating the utility of differentially-private data mining techniques. Whereas evaluation typically ends with a comparison of simple predictive accuracy against non-private methods, we actually simulate the application of a privacy-preserving technique to its domain-specific task, and compare the

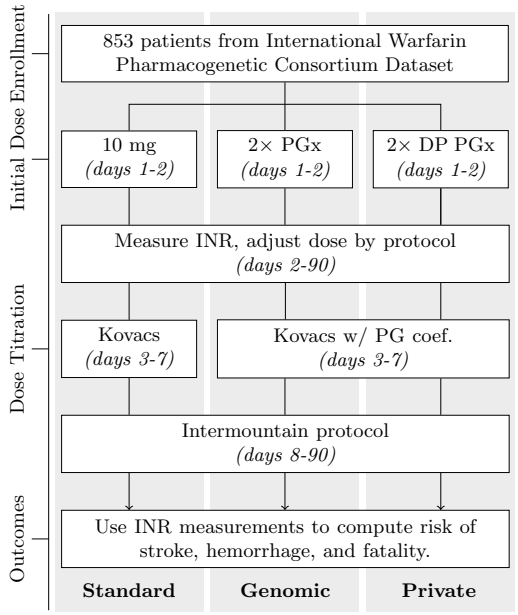


Figure 5: Overview of the Clinical Trial Simulation. *PGx* signifies the pharmacogenomic dosing algorithm, and *DP* differential privacy. The trial consists of three arms differing primarily on initial dosing strategy, and proceeds for 90 days. Details of Kovacs and Intermountain protocol are given in Section 5.3.

outcomes of that task to those achieved without the use of private mechanisms.

5.1 Overview

In order to evaluate the consequences of private genomic dosing algorithms, we simulate a clinical trial designed to measure the effectiveness of new medication regimens. The practice of simulating clinical trials is well-known in the medical research literature [4, 14, 18, 19], where it is used to estimate the impact of various decisions before initiating a costly real-world trial involving human subjects. Our clinical trial simulation follows the design of the CoumaGen clinical trials for evaluating the efficacy of pharmacogenomic warfarin dosing algorithms [3], which is the largest completed real-world clinical trial to date for evaluating these algorithms. At a high level, we train a pharmacogenomic warfarin dosing algorithm and a set of private pharmacogenomic dosing algorithms on the training set. The simulated trial draws random patient samples from the validation set, and for each patient, applies three dosing algorithms to determine the simulated patient's starting dose: the current standard clinical algorithm, the non-private pharmacogenomic algorithm, and one of the private pharmacogenomic algorithms. We then simulate the patient's physiological respose to the doses given by each algorithm using a dose *titration* (i.e., modification) protocol defined by the original CoumaGen trial.

In more detail, our trial simulation defines three parallel *arms* (see Figure 5), each corresponding to a distinct method for assigning the patient's initial dose of warfarin:

1. *Standard:* the current standard practice of initially prescribing a fixed 10mg/day dose.
2. *Genomic:* Use of a genomic algorithm to assign the initial dose.
3. *Private:* Use of a differentially-private genomic algorithm to assign initial dose.

Within each arm, the trial proceeds for 90 simulated days in several stages, as depicted in Figure 5:

1. *Enrollment:* A patient is sampled from the population distribution, and their genotype and demographic characteristics are used to construct an instance of a *Pharmacokinetic/Pharmacodynamic (PK/PD) Model* that characterizes relevant aspects of their physiological response to warfarin (i.e., INR). The PK/PD model contains random variables that are parameterized by genotype and demographic information, and are designed to capture the variance observed in previous population-wide studies of physiological response to warfarin [16].
2. *Initial Dosing:* Depending on which arm of the trial the current patient is in, an initial dose of warfarin is prescribed and administered for the first two days of the trial.
3. *Dose Titration:* For the remaining 88 days of the simulated trial, the patient administers a prescribed dose every 24 hours. At regular intervals specified by the titration protocol, the patient makes "clinic visits" where INR response to previous doses is measured, a new dose is prescribed based on the measured response, and the next clinic visit is scheduled based on the patient's INR and current dose. This is explained in more detail in Sections 5.3 and 5.4.
4. *Measure Outcomes:* The measured responses for each patient at each clinic visit are tabulated, and the risk of negative outcomes is computed.

5.2 Pharmacogenomic Warfarin Dosing

To build the non-private regression model, we use regularized least-squares regression in R, and obtained 15.9% average absolute error (see Figure 6). To build

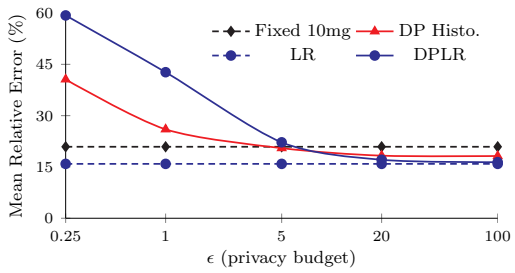


Figure 6: Pharmacogenomic warfarin dosing algorithm performance measured against clinically-deduced ground truth in IWPC dataset.

differentially-private models, we use two techniques: the functional mechanism of Zhang *et al.* [47] and regression models trained on Vinterbo's private projected histograms [44].

To obtain a baseline estimate of these algorithms' performance, we constructed a set of regression models for various privacy budget settings ($\varepsilon = 0.25, 1, 5, 20, 100$) using each of the above methods. The average absolute predictive error, over 100 distinct models at each parameter level, is shown in Figure 6. Although the average error of the private algorithms at low privacy budget settings is quite high, it is not clear how that will affect our simulated patients. In addition to the magnitude of the error, its *direction* (i.e., whether it under- or over-prescribes) matters for different types of risk. Futhermore, because the patient's initial dose is subsequently titrated to more appropriate values according to their INR response, it may be the case that a poor guess for initial dose, as long as the error is not too significant, will only pose a risk during the early portion of the patient's therapy, and a negligible risk overall. Lastly, the accuracy of the standard clinical and non-private pharmacogenomic algorithms are moderate (~15% and 21% error, respectively), and these are the best known methods for predicting initial dose. The difference in accuracy between these and the private algorithm is not extreme (e.g., greater than an order of magnitude), so lacking further information about the correlation between initial dose accuracy and patient outcomes, it is necessary to study their use in greater detail. Removing this uncertainty is the goal of our simulation-based evaluation.

5.3 Dose Assignment and Titration

To assign initial doses and control the titration process in our simulation, we follow the protocol used by the CoumaGen clinical trials on pharmacogenomic warfarin dosing algorithms [3]. In the standard arm, patients are given 10-mg doses on days 1 and 2, followed by dose adjustment according to the Kovacs protocol [29] for days 3 to 7, and final adjustment according to the Intermountain Healthcare protocol [3] for days 8 to 90. Both the Kovacs and Intermountain protocols assign a dose and next appointment time based on the patient's current INR, and possibly their previous dose.

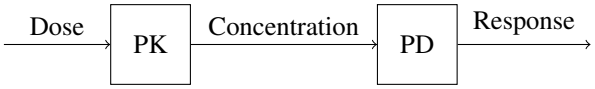


Figure 7: Basic functionality of PK/PD modelling.

The genomic arm differs from the standard arm for days 1-7. The initial dose for days 1-2 is predicted by the pharmacogenomic regression model, and multiplied by 2 [3]. On days 3-7, the Kovacs protocol is used, but the prescribed dose is multiplied by a coefficient C_{pg} that measures the ratio of the predicted pharmacogenomic dose to the standard 10mg initial dose: $C_{\text{pg}} = (\textit{Initial Pharmacogenomic Dose})/(5\text{ mg})$. On days 8-90, the genomic arm proceeds identically to the standard arm. The private arm is identical to the genomic arm, but the pharmacogenomic regression model is replaced with a differentially-private model.

To simulate realistic dosing increments, we assume any combination of three pills from those available at most pharmacies: 0.5, 1, 2, 2.5, 3, 4, 5, 6, 7, and 7.5 mg. The maximum dose was set to 15 mg/day, with possible dose combinations ranging from 0 to 15 mg in 0.5 mg increments.

5.4 PK/PD Model for INR response to Warfarin

A PK/PD model integrates two distinct pharmacologic models—pharmacokinetic (PK) and pharmacodynamic (PD)—into a single set of mathematical expressions that predict the intensity of a subject's response to drug administration over time. *Pharmacokinetics* is the course of drug absorption, distribution, metabolism, and excretion over time. Mechanistically, the pharmacokinetic component of a PK/PD model predicts the *concentration* of a drug in certain parts of the body over time. *Pharmacodynamics* refers to the effect that a drug has on the body, given its concentration at a particular site. This includes the intensity of its therapeutic and toxic effects, which is the role of the pharmacodynamic component of the PK/PD model. Conceptually, these pieces fit together as shown in Figure 7: the PK model takes a sequences of doses, produces a prediction of drug concentraation, which is given to the PD model. The final output is the predicted PD response to the given sequence of doses, both measures being taken over time. The input/output behavior of the model's components can be described as

the following related functions:

$$\begin{aligned} \mathsf{PKPDModel}(genotype, demographics) &\mapsto F_{\text{inr}} \\ F_{\text{inr}}(doses, time) &\mapsto INR \end{aligned}$$

The function PKPDModel transforms a set of patient characteristics, including the relevant genotype and demographic information, into an INR-response predictor F_{inr}. $F_{\text{inr}}(doses, t)$ transforms a sequence of doses, assumed to have been administered at 24-hour intervals starting at $time = 0$, as well as a time t, and produces a prediction of the patient's INR at time t. The function PKPDModel can be thought of as the routine that initializes the parameters in the PK and PD models, and F_{inr} as the function that composes the initialized models to translate dose schedules into INR measurements. For further details of the PK/PD model, consult Appendix A.

5.5 Calculating Patient Risk

INR levels correspond to the coagulation tendency of blood, and thus to the risk of adverse events. Sorensen *et al.* performed a pooled analysis of the correlation between stroke and bleeding events for patients undergoing warfarin treatment at varying INR levels [41]. We use the probabilities for various events as reported in their analysis. We calculate each simulated patient's risk for stroke, intra-cranial hemorrhage, extra-cranial hemorrhage, and fatality based on the predicted INR levels produced by the PK/PD model. At each 24-hour interval, we calculated INR and the corresponding risk for these events. The sum total risk for each event across the entire trial period is the endpoint we use to compare the arms. We also calculated the mean *time in therapeutic range* (TTR) of patients' INR response for each arm. We define TTR as any INR reading between 1.8–3.2, to maintain consistency with previous studies [3, 14].

The results are presented in Figure 8 in terms of relative risk (defined as the quotient of the patient's risk for a certain outcome when using a particular algorithm versus the fixed dose algorithm). The results are striking: for reasonable privacy budgets ($\varepsilon \leq 5$), private pharmacogenomic dosing results in greater risk for stroke, bleeding, and fatality events as compared to the fixed dose protocol. The increased risk is statistically significant for both private algorithms up to $\varepsilon = 5$ and all types of risk (including reduced TTR), except for private histograms, for which there was no significant increase in bleeding events with $\varepsilon > 1$.

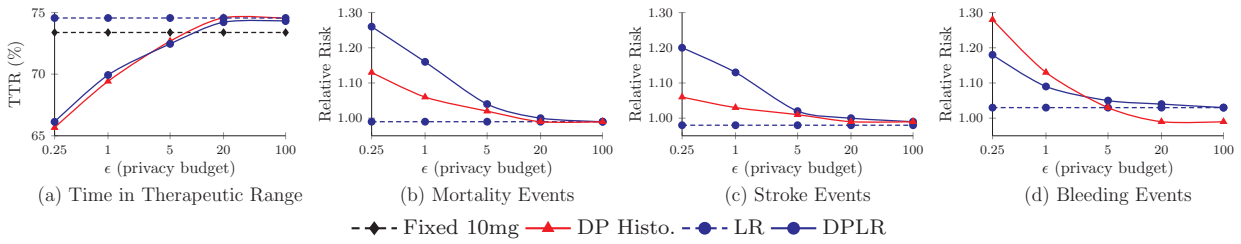
On the positive side, there is evidence that both algorithms may reduce all types of risk at certain privacy levels. Differentially-private histograms performed slightly better, improvements in all types of risk at $\varepsilon \geq 20$. Private linear regression seems to yield lower risk of stroke and fatality and increased TTR at $\varepsilon \geq 20$. However, the difference in bleeding risk for DPLR was not statistically significant at any $\varepsilon \geq 20$. *These results lead us to conclude that there is evidence that differentially-private statistical models may provide effective algorithms for predicting initial warfarin dose, but only at low settings of* $\varepsilon \geq 20$ *that yield little privacy (see Section 4).*

6 Related Work

The tension between privacy and data utility has been explored by several authors. Brickell and Shmatikov [6] found strong evidence for a tradeoff in attribute privacy and predictive performance in common data mining tasks when k-anonymity, ℓ-diversity, and t-closeness are applied before releasing a full dataset. Differential privacy arose partially as a response to Dalenius' desideratum: *anything that can be learned from the database about a specific individual should be learnable without access to the database* [9]. Dwork showed the impossibility of achieving this result in the presence of utility requirements [11], and proposed an alternative goal that proved feasible to achieve in many settings: *the risk to one's privacy should not substantially increase as a result of participating in a statistical database*. Differential privacy formalizes this goal, and constructive research on the topic has subsequently flourised.

Differential privacy is often misunderstood by those who wish to apply it, as pointed out by Dwork and others [13]. Kifer and Machanavajjhala [25] addressed several common misconceptions about the topic, and showed that under certain conditions, it fails to achieve a privacy goal related to Dwork's: *nearly all evidence of an individual's participation should be removed*. Using hypothetical examples from social networking and census data release, they demonstrate that when rows in a database are correlated, or when previous exact statistics for a dataset have been released, this notion of privacy may be violated even when differential privacy is used. Part of our work extends theirs by giving a concrete examples from a realistic application where common misconceptions about differential privacy lead to surprising privacy breaches, i.e., that it will protect genomic attributes from unwanted disclosure. We further extend their analysis by providing a quantitative study of the tradeoff between privacy and utility in the application.

Others have studied the degree to which differential privacy leaks various types of information. Cormode showed that if one is allowed to pose certain queries relating sensitive attributes to quasi-identifiers, it is possible to build a differentially-private Naive Bayes classifier that accurately predicts the sensitive attribute [8]. In contrast, we show that given a model for predicting a



(a) Time in Therapeutic Range (b) Mortality Events (c) Stroke Events (d) Bleeding Events

Figure 8: Trial outcomes for fixed dose, non-private linear regression (LR), differentially-private linear regression (DPLR), and private histograms. Horizontal axes represent ε.

certain outcome from a set of inputs (and no control over the queries used to construct the model), it is possible to make accurate predictions in the *reverse* direction: predict one of the inputs given a subset of the other values. Lee and Clifton [30] recognize the problem of setting ε and its relationship to the *relative* nature of differential privacy, and later [31] propose an alternative parametization of differential privacy in terms of the *probability that an individual contributes to the resulting model.* While this may make the privacy guarantee easier for non-specialists to understand, its close relationship to the standard definition suggests that it may not be effective at mitigating the types of disclosures documented in this paper; evaluating its efficacy remains future work, as we are not aware of any existing implementations that support their definition.

The risk of sensitive information disclosure in medical studies has been examined by many. Wang *et al.* [46], Homer *et al.* [20] and Sankararaman *et al.* [39] show that it is possible to recover parts of an individual's genotype given partial genetic information and detailed statistics from a GWAS. They do not evaluate the efficacy of their techniques against private versions of the statistics, and do not consider the problem of inference from a model derived from the statistics. Sweeny showed that a few pieces of identifying information are suitable to identify patients in medical records [42]. Loukides *et al.* [34] show that it is possible to identify a wide range of sensitive patient information from de-identified clinical data presented in a form standard among medical researchers, and later proposed a domain-specific utility-preserving scheme similar to *k*-anonymity for mitigating these breaches [35]. Dankar and Emam [10] discuss the use of differential privacy in medical applications, pointing out the various tradeoffs between interactive and non-interactive mechanisms and the limitation of utility guarantees in differential privacy, but do not study its use in any specific medical applications.

Komarova *et al.* [28] present an in-depth study of the problem of *partial disclosure*. There is some similarity between the model inversion attacks discussed here and this notion of partial disclosure. One key difference is that in the case of model inversion, an adversary is given the actual function corresponding to a statistical estimator (e.g., a linear model in our case study), whereas Komarova *et al.* consider static estimates from combined public and private sources. In the future we will investigate whether the techniques described by Komarova *et al.* can be used to refine, or provide additional information for, model inversion attacks.

7 Conclusion

We conducted the first end-to-end case study of the use of differential privacy in a medical application, exploring the tradeoff between privacy and utility that occurs when existing differentially-private algorithms are used to guide dosage levels in warfarin therapy. Using a new technique called *model inversion*, we repurpose pharmacogenetic models to infer patient genotype. We showed that models used in warfarin therapy introduce a threat to patients' genomic privacy. When models are produced using state-of-the-art differential privacy mechanisms, genomic privacy is protected for small $\varepsilon(\leq 1)$, but as ε increases towards larger values this protection vanishes.

We evaluated the *utility* of differential privacy mechanisms by simulating clinical trials that use private models in warfarin therapy. This type of evaluation goes beyond what is typical in the literature on differential privacy, where raw statistical accuracy is the most common metric for evaluating utility. We show that differential privacy substantially interferes with the main purpose of these models in personalized medicine: for ε values that protect genomic privacy, which is the central privacy concern in our application, the risk of negative patient outcomes increases beyond acceptable levels.

Our work provides a framework for assessing the tradeoff between privacy and utility for differential privacy mechanisms in a way that is directly meaningful for specific applications. For settings in which certain levels of utility performance *must* be achieved, and this tradeoff cannot be balanced, then alternative means of protecting individual privacy must be employed.

References

[1] Clarification of optimal anticoagulation through genetics. `http://coagstudy.org`.

[2] The pharmacogenomics knowledge base. `http://www.pharmgkb.org`.

[3] J. L. Anderson, B. D. Horne, S. M. Stevens, A. S. Grove, S. Barton, Z. P. Nicholas, S. F. Kahn, H. T. May, K. M. Samuelson, J. B. Muhlestein, J. F. Carlquist, and for the Couma-Gen Investigators. Randomized trial of genotype-guided versus standard warfarin dosing in patients initiating oral anticoagulation. *Circulation*, 116(22):2563–2570, 2007.

[4] P. L. Bonate. Clinical trial simulation in drug development. *Pharmaceutical Research*, 17(3):252–256, 2000.

[5] L. D. Brace. Current status of the international normalized ratio. *Lab Medicine*, 32(7):390–392, 2001.

[6] J. Brickell and V. Shmatikov. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *KDD*, 2008.

[7] J. Carlquist, B. Horne, J. Muhlestein, D. Lapp, B. Whiting, M. Kolek, J. Clarke, B. James, and J. Anderson. Genotypes of the Cytochrome P450 Isoform, CYP2C9, and the Vitamin K Epoxide Reductase Complex Subunit 1 conjointly determine stable warfarin dose: a prospective study. *Journal of Thrombosis and Thrombolysis*, 22(3), 2006.

[8] G. Cormode. Personal privacy vs population privacy: learning to attack anonymization. In *KDD*, 2011.

[9] T. Dalenius. Towards a methodology for statistical disclosure control. *Statistik Tidskrift*, 15(429-444):2–1, 1977.

[10] F. K. Dankar and K. El Emam. The application of differential privacy to health data. In *ICDT*, 2012.

[11] C. Dwork. Differential privacy. In *ICALP*. Springer, 2006.

[12] C. Dwork. The promise of differential privacy: A tutorial on algorithmic techniques. In *FOCS*, 2011.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Differential privacy: A primer for the perplexed. In *Joint UNECE/Eurostat work session on statistical data confidentiality*, 2011.

[14] V. A. Fusaro, P. Patil, C.-L. Chi, C. F. Contant, and P. J. Tonellato. A systems approach to designing effective clinical trials using simulations. *Circulation*, 127(4):517–526, 2013.

[15] S. R. Ganta, S. P. Kasiviswanathan, and A. Smith. Composition attacks and auxiliary information in data privacy. In *KDD*, 2008.

[16] A. K. Hamberg, Dahl, M. L., M. Barban, M. G. Srordo, M. Wadelius, V. Pengo, R. Padrini, and E. Jonsson. A PK-PD model for predicting the impact of age, CYP2C9, and VKORC1 genotype on individualization of warfarin therapy. *Clinical Pharmacology Theory*, 81(4):529–538, 2007.

[17] D. Hand and R. Till. A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine Learning*, 45(2):171–186, 2001.

[18] N. Holford, S. C. Ma, and B. A. Ploeger. Clinical trial simulation: A review. *Clinical Pharmacology Theory*, 88(2):166–182.

[19] N. H. G. Holford, H. C. Kimko, J. P. R. Monteleone, and C. C. Peck. Simulation of clinical trials. *Annual Review of Pharmacology and Toxicology*, 40(1):209–234, 2000.

[20] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genetics*, 4(8), 08 2008.

[21] International Warfarin Pharmacogenetic Consortium. Estimation of the warfarin dose with clinical and pharmacogenetic data. *New England Journal of Medicine*, 360(8):753–764, 2009.

[22] E. Jaynes. On the rationale of maximum-entropy methods. *Proceedings of the IEEE*, 70(9), Sept 1982.

[23] F. Kamali and H. Wynne. Pharmacogenetics of warfarin. *Annual Review of Medicine*, 61(1):63–75, 2010.

[24] S. P. Kasiviswanathan, M. Rudelson, and A. Smith. The power of linear reconstruction attacks. In *SODA*, 2013.

[25] D. Kifer and A. Machanavajjhala. No free lunch in data privacy. In *SIGMOD*, 2011.

[26] M. J. Kim, S. M. Huang, U. A. Meyer, A. Rahman, and L. J. Lesko. A regulatory science perspective on warfarin therapy: a pharmacogenetic opportunity. *J Clin Pharmacol*, 49:138–146, Feb 2009.

[27] S. E. Kimmel, B. French, S. E. Kasner, J. A. Johnson, J. L. Anderson, B. F. Gage, Y. D. Rosenberg, C. S. Eby, R. A. Madigan, R. B. McBane, S. Z. Abdel-Rahman, S. M. Stevens, S. Yale, E. R. Mohler, M. C. Fang, V. Shah, R. B. Horenstein, N. A. Limdi, J. A. Muldowney, J. Gujral, P. Delafontaine, R. J. Desnick, T. L. Ortel, H. H. Billett, R. C. Pendleton, N. L. Geller, J. L. Halperin, S. Z. Goldhaber, M. D. Caldwell, R. M. Califf, and J. H. Ellenberg. A pharmacogenetic versus a clinical algorithm for warfarin dosing. *New England Journal of Medicine*, 369(24):2283–2293, 2013. PMID: 24251361.

[28] T. Komarova, D. Nekipelov, and E. Yakovlev. *Estimation of Treatment Effects from Combined Data: Identification versus Data Security*. NBER volume Economics of Digitization: An Agenda, To appear.

[29] M. J. Kovacs, M. Rodger, D. R. Anderson, B. Morrow, G. Kells, J. Kovacs, E. Boyle, and P. S. Wells. Comparison of 10-mg and 5-mg warfarin initiation nomograms together with low-molecular-weight heparin for outpatient treatment of acute venous thromboembolism. *Annals of Internal Medicine*, 138(9):714–719, 2003.

[30] J. Lee and C. Clifton. How much is enough? Choosing ε for differential privacy. In *ISC*, 2011.

[31] J. Lee and C. Clifton. Differential identifiability. In *KDD*, 2012.

[32] J. Lei. Differentially private m-estimators. In *NIPS*, 2011.

[33] Y. Lindell and E. Omri. A practical application of differential privacy to personalized online advertising. *IACR Cryptology ePrint Archive*, 2011.

[34] G. Loukides, J. C. Denny, and B. Malin. The disclosure of diagnosis codes can breach research participants' privacy. *Journal of the American Medical Informatics Association*, 17(3):322–327, 2010.

[35] G. Loukides, A. Gkoulalas-Divanis, and B. Malin. Anonymization of electronic medical records for validating genome-wide association studies. *Proceedings of the National Academy of Sciences*, 107(17):7898–7903, Apr. 2010.

[36] A. Narayanan and V. Shmatikov. Robust deanonymization of large sparse datasets. In *Oakland*, 2008.

[37] A. Narayanan and V. Shmatikov. Myths and fallacies of *Personally Identifiable Information*. *Commun. ACM*, 53(6), June 2010.

[38] J. Reed, A. J. Aviv, D. Wagner, A. Haeberlen, B. C. Pierce, and J. M. Smith. Differential privacy for collaborative security. In *Proceedings of the Third European Workshop on System Security*, EUROSEC, 2010.

[39] S. Sankararaman, G. Obozinski, M. I. Jordan, and E. Halperin. Genomic privacy and limits of individual detection in a pool. *Nature Genetics*, 41(9):965–967, 2009.

[40] E. A. Sconce, T. I. Khan, H. A. Wynne, P. Avery, L. Monkhouse, B. P. King, P. Wood, P. Kesteven, A. K. Daly, and F. Kamali. The impact of CYP2C9 and VKORC1 genetic polymorphism and patient characteristics upon warfarin dose requirements: proposal for a new dosing regimen. *Blood*, 106(7):2329–2333, 2005.

[41] S. V. Sorensen, S. Dewilde, D. E. Singer, S. Z. Goldhaber, B. U. Monz, and J. M. Plumb. Cost-effectiveness of warfarin: Trial versus real-world stroke prevention in atrial fibrillation. *American Heart Journal*, 157(6):1064 – 1073, 2009.

[42] L. Sweeney. Simple demographics often identify people uniquely. 2000.

[43] F. Takeuchi, R. McGinnis, S. Bourgeois, C. Barnes, N. Eriksson, N. Soranzo, P. Whittaker, V. Ranganath, V. Kumanduri, W. McLaren, L. Holm, J. Lindh, A. Rane, M. Wadelius, and P. Deloukas. A genome-wide association study confirms VKORC1, CYP2C9, and CYP4F2 as principal genetic determinants of warfarin dose. *PLoS Genet*, 5(3), 03 2009.

[44] S. Vinterbo. Differentially private projected histograms: Construction and use for prediction. In *ECML-PKDD*, 2012.

[45] D. Vu and A. Slavkovic. Differential privacy for clinical trial data: Preliminary evaluations. In *ICDM Workshops*, 2009.

[46] R. Wang, Y. F. Li, X. Wang, H. Tang, and X. Zhou. Learning your identity and disease from research papers: information leaks in genome wide association studies. In *CCS*, 2009.

[47] J. Zhang, Z. Zhang, X. Xiao, Y. Yang, and M. Winslett. Functional mechanism: regression analysis under differential privacy. In *VLDB*, 2012.

A PK/PD Model Details

We adopted a previously-developed PK/PD INR model to predict each patient's INR response to previous dosing choices [16]. The PK component of the model is a *two-compartment* model with *first-order absorption*. A two-compartment model assumes an abstract representation of the body as two discrete sections: the first being a *central* compartment into which a drug is administered and a *peripheral* compartment into which the drug eventually distributes. The central compartment (assumed to have volume V_1) represents tissues that equilibrate rapidly with blood (e.g., liver, kidney, *etc.*), and the peripheral (volume V_2) those that equilibrate slowly (e.g., muscle, fat, *etc.*). Three *rate constants* govern transfer between the compartments and elimination: k_{12}, k_{21}, for the central-peripheral and peripheral-central transfer, and k_{el} for elimination from the body, respectively. V_1, V_2, k_{12}, and k_{21} are related by the following equality: $V_1 k_{12} = V_2 k_{21}$. The *absorption rate* k_a governs the rate at which the drug enters the central compartment. In the model used in our simulation, each of these parameters is represented by a random variable whose distribution has been fit to observed population measurements of Warfarin absorption, distribution, metabolism, and elimination [16]. The elimination-rate constant k_{el} is parameterized by the patient's CYP2C9 genotype.

Given a set of PK parameters, the Warfarin concentration in the central compartment over time is calculated using standard two-compartment PK equations for oral dosing. Concentration in two-compartment pharmacokinetics diminishes in two distinct phases with differing rates: the α ("distribution") phase, and β ("elimination") phase. The expression for concentration C over time assuming doses $D_1, \ldots, D_n$ administered at times $t_{D_1}, \ldots, t_{D_n}$ has another term corresponding to the effect of oral absorption:

$$C(t) = \sum_{i=1}^{n} D_i (A e^{-\alpha t_i} + B e^{-\beta t_i} - (A+B) e^{-k_a t_i})$$

with $t_i = t - t_{D_i}$ and α, β satisfying $\alpha\beta = k_{21} k_{el}$, $\alpha + \beta = k_{el} + k_{12} + k_{21}$, and

$$A = \frac{k_a}{V_1} \frac{k_{21} - \alpha}{(k_a - \alpha)(\beta - \alpha)} \quad B = \frac{k_a}{V_1} \frac{k_{21} - \beta}{(k_a - \beta)(\alpha - \beta)}$$

Our model contains an error term with a zero-centered log-normal distribution whose variance depends on whether or not steady-state dosing has occurred; the term is given in the appendix of Hamberg *et al.* [16].

PD Model The PD model used in our simulations is an *inhibitory sigmiod-$E_{\max}$ model*. Recall that the purpose of the PD model is to describe the physiological response E, in this case INR, to Warfarin concentration at a particular time. $E_{\max}$ represents the maximal response, i.e., the maximal inhibition of coagulation, and E_{50} the concentration of Warfarin producing half-maximal inhibition. $E_{\max}$ is fixed to 1, and E_{50} is a patient-specific random variable that is a function of the patient's VKORC1 genotype. A sigmoidocity factor γ is used to model the fact that the concentration-effect response of Warfarin corresponds to a sigmoid curve at lower concentrations. The basic formula for calculating E at time t from concentration is: $1 - (E_{\max} C(t)^\gamma)/(E_{50}^\gamma + C(t)^\gamma)$. However, Warfarin exhibits a delay between exposure and anticoagulation response. To characterize this feature, Hamberg *et al.* showed that extending the basic $E_{\max}$ model with a *transit compartment model* with two parallel chains is adequate [16], as shown in Figure 9. The delay between exposure and concentration is modeled by assuming that the drug travels along two parallel *compartment chains* of differing lengths and turnover rates. The transit rate between compartments on the two chains is given by two constants k_{tr1} and k_{tr2}. The first chain consists of six compartments, and the second a single compartment. The first transit constant is a random zero-centered log-normal variable, whereas empirical data did not reilably support variance in the second [16]. The amount in a given compartment i, A_i, at time t is described by a system of coupled ordinary differential equations:

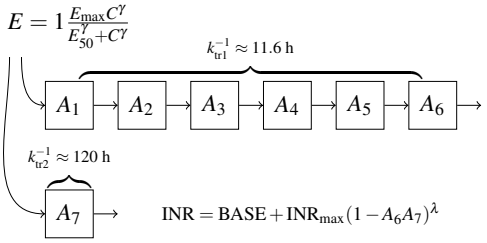


Figure 9: Overview of transit-compartment PD model [16].

$$\frac{dA_1}{dt} = k_{tr1} \left(1 - \frac{E_{\max} C(t)^\gamma}{E_{50}^\gamma + C(t)^\gamma}\right) - k_{tr1} A_1$$

$$\frac{dA_n}{dt} = k_{tr1}(A_{n-1} - A_n), n = 2,3,4,5,6$$

$$\frac{dA_7}{dt} = k_{tr2} \left(1 - \frac{E_{\max} C(t)^\gamma}{E_{50}^\gamma + C(t)^\gamma}\right) - k_{tr2} A_7$$

The final expression for INR at time t is given by solving for A_7 and A_7 starting from initial conditions $A_i = 1$, and calculating the expression: $\log(\text{INR}) = \log(\text{Base} + \text{INR}_{\max}(1 - A_6 A_7)^\lambda) + \varepsilon_{\text{INR}}$. In this expression, Base is the patient's baseline INR, $\text{INR}_{\max}$ is the maximal INR (assumed to be 20 [16]), λ is a scaling factor derived from empirical data [16], and ε_{INR} is a zero-centerd, symmetrically-distributed random variable with variance determined from empirical data [16].

Mimesis Aegis: A Mimicry Privacy Shield

A System's Approach to Data Privacy on Public Cloud

Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332

{billy, pchung, csong84, yeongjin.jang, wenke, sasha.boldyreva}@cc.gatech.edu

Abstract

Users are increasingly storing, accessing, and exchanging data through public cloud services such as those provided by Google, Facebook, Apple, and Microsoft. Although users may want to have faith in cloud providers to provide good security protection, the confidentiality of any data in public clouds can be violated, and consequently, while providers may not be "doing evil," we can not and should not trust them with data confidentiality.

To better protect the privacy of user data stored in the cloud, in this paper we propose a privacy-preserving system called *Mimesis Aegis (M-Aegis)* that is suitable for mobile platforms. M-Aegis is a new approach to user data privacy that not only provides isolation but also preserves the user experience through the creation of a conceptual layer called *Layer 7.5 (L-7.5)*, which is interposed between the application (OSI Layer 7) and the user (Layer 8). This approach allows M-Aegis to implement true end-to-end encryption of user data with three goals in mind: 1) complete data and logic isolation from untrusted entities; 2) the preservation of original user experience with target apps; and 3) applicable to a large number of apps and resilient to app updates.

In order to preserve the exact application workflow and look-and-feel, M-Aegis uses L-7.5 to put a transparent window on top of existing application GUIs to both intercept plaintext user input before transforming the input and feeding it to the underlying app, and to reverse-transform the output data from the app before displaying the plaintext data to the user. This technique allows M-Aegis to transparently integrate with most cloud services without hindering usability and without the need for reverse engineering. We implemented a prototype of M-Aegis on Android and show that it can support a number of popular cloud services, e.g. Gmail, Facebook Messenger, WhatsApp, etc.

Our performance evaluation and user study show that users incur minimal overhead when adopting M-Aegis on Android: imperceptible encryption/decryption latency and a low and adjustable false positive rate when searching over encrypted data.

1 Introduction

A continuously increasing number of users now utilize mobile devices [2] to interact with public cloud services (PCS) (e.g. Gmail, Outlook, and WhatsApp) as an essential part of their daily lives. While the user's connectivity to the Internet is improved with mobile platforms, the problem of preserving data privacy while interacting with PCS remains unsolved. In fact, news about the US government's alleged surveillance programs reminds everybody about a very unsatisfactory status quo: while PCS are essentially part of everyday life, the default method of utilizing them exposes users to privacy breaches, because it implicitly requires the users to trust the PCS providers with the confidentiality of their data; but such trust is unjustified, if not misplaced. Incidents that demonstrate breach of this trust are easy to come by: 1) PCS providers are bound by law to share their users' data with surveillance agencies [14], 2) it is the business model of the PCS providers to mine their users' data and share it with third parties [11, 22, 24, 40], 3) operator errors [34] can result in unintended data access, and 4) data servers can be compromised by attackers [47].

To alter this undesirable status quo, solutions should be built based on an updated trust model of everyday communication that better reflects the reality of the threats mentioned above. In particular, new solutions must first assume PCS providers to be untrusted. This implies that all other entities that are controlled by the PCS providers, including the apps that users installed to engage with the PCS, must also be assumed untrusted.

Although there are a plethora of apps available today that come in various combinations of look and feel and features, we observed that many of these apps provide text communication services (e.g. email or private/group

messaging categories). Users can still enjoy the same quality of service[1] without needing to reveal their plaintext data to PCS providers. PCS providers are essentially message routers that can function normally without needing to know the content of the messages being delivered, analogous to postmen delivering letters without needing to learn the actual content of the letters.

Therefore, applying end-to-end encryption (E2EE) without assuming trust in the PCS providers seems to solve the problem. However, in practice, the direct application of E2EE solutions onto the mobile device environment is more challenging than originally thought [65, 59]. A good solution must present clear advantages to the entire mobile security ecosystem. In particular it must account for these factors: 1) the users' ease-of-use, hence acceptability and adoptability; 2) the developers' efforts to maintain support; and 3) the feasibility and deployability of solution on a mobile system. From this analysis, we formulate three design goals that must be addressed coherently:

1. For a solution to be secure, it must be properly isolated from untrusted entities. It is obvious that E2EE cannot protect data confidentiality if plaintext data or an encryption key can be compromised by architectures that risk exposing these values. Traditional solutions like PGP [15] and newer solutions like Gibberbot [5], TextSecure [12], and SafeSlinger [41] provide good isolation, but force users to use custom apps, which can cause usability problems (refer to (2)). Solutions that repackage/rewrite existing apps to introduce additional security checks [68, 26] do not have this property (further discussed in Sect. 2.3). Solutions in the form of browser plugins/extensions also do not have this property (further discussed in Sect. 2.2), and they generally do not fit into the mobile security landscape because many mobile browsers do not support extensions [7], and mobile device users do not favor using mobile browsers [27] to access PCS. Therefore, we rule out conventional browser-plugin/extension-based solutions.
2. For a solution to be adoptable, it must preserve the user experience. We argue that users will not accept solutions that require them to switch between different apps to perform their daily tasks. Therefore, simply porting solutions like PGP to a mobile platform would not work, because it forces users to use a separate and custom app, and it is impossible to recreate the richness and unique user experience of all existing text routing apps offered by various PCS providers today. In the context of mobile devices, PCS are competing for market share not only by offering more reliable infrastructure to facilitate user communication, but also by offering a better user experience [16, 58]. Ultimately, users will choose apps that feel the most comfortable. To reduce interference with a user's interaction with the app of their choice, security solutions must be retrofittable to existing apps. Solutions that repackage/rewrite existing apps have this criterion.
3. For a solution to be sustainable, it must be easy to maintain and scalable: the solution must be sufficiently general-purpose, require minimal effort to support new apps, and withstand app updates. In the past, email was one of the very few means of communication. Protecting it is relatively straightforward because email protocols (e.g. POP and IMAP) are well defined. Custom privacy-preserving apps can therefore be built to serve this need. However, with the introduction of PCS that are becoming indispensable in a user's everyday life, a good solution should also be able to integrate security features into apps without requiring reverse engineering of the apps' logic and/or network protocols, which are largely undocumented and possibly proprietary (e.g. Skype, WhatsApp, etc.).

In this paper, we introduce *Mimesis Aegis (M-Aegis)*, a privacy-preserving system that mimics the look and feel of existing apps to preserve their user experience and workflow on mobile devices, without changing the underlying OS or modifying/repackaging existing apps. M-Aegis achieves the three design goals by operating at a conceptual layer we call *Layer 7.5 (L-7.5)* that is positioned above the existing application layer (OSI Layer 7 [8]), and interacts directly with the user (popularly labeled as Layer 8 [19, 4]).

From a system's perspective, L-7.5 is a transparent window in an isolated process that interposes itself between Layer 7 and 8. The interconnectivity between these layers is achieved using the accessibility framework, which is available as an essential feature on modern operating systems. Note that utilizing accessibility features for unorthodox purposes have been proposed by prior work [56, 48] that achieves different goals. L-7.5 extracts the GUI information of an app below it through the OS's user interface automation/accessibility (UIA) library. Using this information, M-Aegis is then able to proxy user input by rendering its own GUI (with a different color as visual cue) and subsequently handle those input (e.g. to process plaintext data or intercept user button click). Using the same UIA library, L-7.5 can also programmatically interact with various UI components of the app below on behalf of the user (refer to Sect. 3.3.2 for more details). Since major software vendors today have pledged their commitment towards continuous sup-

[1] the apps' functionalities and user experience are preserved

port and enhancement of accessibility interface for developers [9, 20, 6, 1], our UIA-based technique is applicable and sustainable on all major platforms.

From a security design perspective, M-Aegis provides two privacy guarantees during a user's interaction with a target app: 1) all input from the user first goes to L-7.5 (and is optionally processed) before being passed to an app. This means that confidential data and user intent can be fully captured; and 2) all output from the app must go through L-7.5 (and is optionally processed) before being displayed to the user.

From a developer's perspective, accessing and interacting with a target app's UI components at L-7.5 is similar to that of manipulating the DOM tree of a web app using JavaScript. While DOM tree manipulation only works for browsers, UIA works for all apps on a platform. To track the GUI of an app, M-Aegis relies on resource id names available through the UIA library. Therefore, M-Aegis is resilient to updates that change the look and feel of the app (e.g. GUI position or color). It only requires resource id names to remain the same, which, through empirical evidence, often holds true. Even if a resource id changes, minimal effort is required to rediscover resource id names and remap them to the logic in M-Aegis. From our experience, M-Aegis does not require developer attention across minor app updates.

From a user's perspective, M-Aegis is visible as an always-on-top button. When it is turned on, users will perceive that they are interacting with the original app in plaintext mode. The only difference is the GUI of the original app will appear in a different color to indicate that protection is activated. This means that subtle features that contribute towards the entire user experience such as spell checking and in-app navigation are also preserved. However, despite user perception, the original app *never* receives plaintext data. Figure 1 gives a high level idea of how M-Aegis creates an L-7.5 to protect user's data privacy when interacting with Gmail.

For users who would like to protect their email communications, they will also be concerned if encryption will affect their ability to search, as it is an important aspect of user productivity [64]. For this purpose, we designed and incorporated a new searchable encryption scheme named *easily-deployable efficiently-searchable symmetric encryption scheme* (EDESE) into M-Aegis that allows search over encrypted content without any server-side modification. We briefly discuss the design considerations and security concerns involved in supporting this functionality in Sect 3.3.4.

As a proof of concept, we implemented a prototype M-Aegis on Android that protects user data when interfacing with text-based PCS. M-Aegis supports email apps like Gmail and messenger apps like Google Hangout,

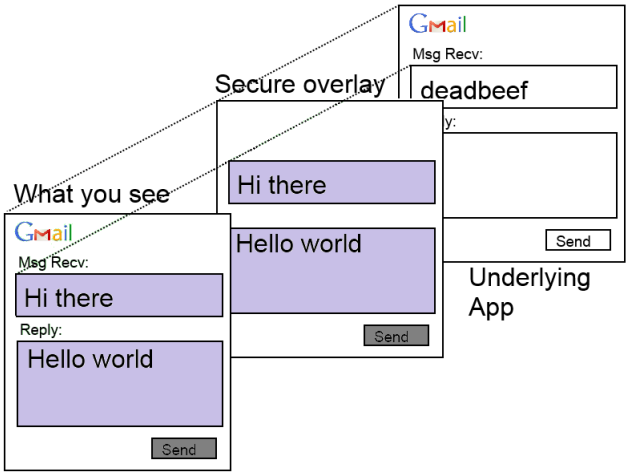


Figure 1: This diagram shows how M-Aegis uses L-7.5 to transparently reverse-transform the message "deadbeef" into "Hi there", and also allows a user to enter their plaintext message "Hello world" into M-Aegis's text box. To the user, the GUI looks exactly the same as the original app. When the user decides to send a message, the "Hello world" message will be transformed and relayed to the underlying app.

WhatsApp, and Facebook Chat. It protects data privacy by implementing E2EE that passes no plaintext to an app while also preserving the user experience and workflow. We also implemented a version of M-Aegis on the desktop to demonstrate the generality of our approach. Our initial performance evaluation and user study shows that users incur minimal overhead in adopting M-Aegis on Android. There is imperceptible encryption/decryption latency and a low and adjustable false positive rate when searching over encrypted data.

In summary, these are the major contributions of our work:

- We introduced Layer 7.5 (L-7.5), a conceptual layer that directly interacts with users on top of existing apps. This is a novel system approach that provides seemingly contrasting features: transparent interaction with a target app and strong isolation from the target app.
- We designed and built Mimesis Aegis based on the concept of L-7.5, a system that preserves user privacy when interacting with PCS by ensuring data confidentiality. Essential functionalities of existing apps, especially search (even over encrypted data), are also supported without any server-side modification.
- We implemented two prototypes of Mimesis Aegis, one on Android and the other on Windows, with

support for various popular public cloud services, including Gmail, Facebook Messenger, Google Hangout, WhatsApp, and Viber.

- We designed and conducted a user study that demonstrated the acceptability of our solution.

The rest of the paper is structured as follows. Section 2 compares our work to related work. Section 3 discusses the threat model and the design of M-Aegis. Section 4 presents the implementation of M-Aegis and the challenges we solved during the process. Section 5 presents performance evaluations and user study of the acceptability of M-Aegis on Android. Section 6 discusses limitations of our work and answers some common questions that readers may have about our system. Section 7 discusses future work and concludes our work.

2 Related Work

Since M-Aegis is designed to achieve the three design goals described in Sect. 1 while seamlessly integrating end-to-end encryption into user's communication, we discuss how well existing works achieve some of these goals and how they differ from our work. As far as we know, there is no existing work that achieves all the three design goals.

2.1 Standalone Solutions

There are many standalone solutions that aim to protect user data confidentiality. Solutions like PGP [15] (including S/MIME [37]), Gibberbot [5], TextSecure [12], SafeSlinger [41], and FlyByNight [55] provides secure messaging and/or file transfer through encryption of user data. These solutions provide good isolation from untrusted entities. However, since they are designed as standalone custom apps, they do not preserve the user experience, requiring users to adopt a new workflow on a custom app. More importantly, these solutions are not retrofittable to existing apps on the mobile platform.

Like M-Aegis, Cryptons [36] introduced a similarly strong notion of isolation through its custom abstractions. However, Cryptons assumes a completely different threat model that trusts PCS, and requires both server and client (app) modifications. Thus, Cryptons could not protect a user's communication using existing messaging apps while assuming the provider to be untrusted. We also argue that it is non-trivial to modify Cryptons to achieve the three design goals we mentioned in Sect. 1.

2.2 Browser Plugin/Extension Solutions

Other solutions that focus on protecting user privacy include Cryptocat [3], Scramble! [24], TrustSplit [40], NOYB (None of Your Business) [46], and SafeButton [53]. Some of these assume different threat models, and achieve different goals. For example, NOYB protects a user's Facebook profile data while SafeButton tries to keep a user's browsing history private. Most of these solutions try to be transparently integrated into user workflow. However, since these solutions are mostly based on browser plugins/extensions, they are not applicable to the mobile platform.

Additionally, Cryptocat and TrustSplit require new and/or independent service providers to support their functionalities. However, M-Aegis works with the existing service providers without assuming trust or requiring modification to server-side communication.

2.3 Repackaging/Rewriting Solutions

There is a category of work that repackages/rewrites an app's binary to introduce security features, such as Aurasium [68], Dr. Android [49], and others [26]. Our solution is similar to these approaches in that we can retrofit our solutions to existing apps and still preserve user experience, but is different in that M-Aegis' coverage is not limited to apps that do not contain native code. Also, repackaging-based approaches suffer from the problem that they will break app updates. In some cases, the security of such solutions can be circumvented because the isolation model is unclear, i.e. the untrusted code resides in the same address space as the reference monitor (e.g. Aurasium).

2.4 Orthogonal Work

Although our work focuses on user interaction on mobile platforms with cloud providers, we assume a very different threat model than those that focus on more robust permission model infrastructures and those that focus on controlling/tracking information flow, such as TaintDroid [38] and Airbag [67]. These solutions require changes to the underlying app, framework, or the OS, but M-Aegis does not.

Access Control Gadgets (ACG) [57] uses user input as permission granting intent to allow apps to access user owned resources. Although we made the same assumptions as ACG to capture authentic user input, ACG is designed for a different threat model and security goal than ours. Furthermore, ACG requires a modified kernel but M-Aegis does not.

Persona [23] presents a completely isolated and new online social network that provides certain privacy and security guarantees to the users. While related, it differs from the goal of M-Aegis.

Frientegrity [43] and Gyrus [48] focus on different aspects of integrity protection of a user's data.

Tor [35] is well known for its capability to hide a user's IP address while browsing the Internet. However, it focuses on anonymity guarantees while M-Aegis focuses on data confidentiality guarantees.

Off-the-record messaging (OTR) [30] is a secure communication protocol that provides perfect forward secrecy and malleable encryption. While OTR can be implemented on M-Aegis using the same design architecture to provide these extra properties, it is currently not the focus of our work.

3 System Design

3.1 Design Goals

In this section, we formally reiterate our design goals. We posit that a good solution must:

1. Offer good security by applying strong isolation from untrusted entities (defined in Sect. 3.2).
2. Preserve the user experience by providing users transparent interaction with existing apps.
3. Be easy to maintain and scale by devising a sufficiently general-purpose approach.

Above all, these goals must be satisfied within the unique set of constraints found in the mobile platform, including user experience, transparency, deployability, and adoptability factors.

3.2 Threat Model

3.2.1 In-Scope Threats

We begin with the scope of threats that M-Aegis is designed to protect against. In general, there are three parties that pose threats to the confidentiality of users' data exposed to public cloud through mobile devices. Therefore, we assume these parties to be untrusted in our threat model:

- Public cloud service (PCS) providers. Sensitive data stored in the public cloud can be compromised in several ways: 1) PCS providers can be compelled by law [21] to provide access to a user's sensitive data to law enforcement agencies [14]; 2) the business model of PCS providers creates strong incentive for them to share/sell user data with third parties [11, 22, 24, 40]; 3) PCS administrators who have access to the sensitive data may also compromise the data, either intentionally [14] or not [34]; and 4) vulnerabilities of the PCS can be exploited by attackers to exfiltrate sensitive data [47].
- Client-side apps. Since client-side apps are developed by PCS providers to allow a user to access their services, it follows that these apps are considered untrusted too.
- Middle boxes between a PCS and a client-side app. Sensitive data can also be compromised when it is transferred between a PCS and a client-side app. Incorrect protocol design/implementation may allow attackers to eavesdrop on plaintext data or perform Man-in-the-Middle attacks [39, 18, 13].

M-Aegis addresses the above threats by creating L-7.5, which it uses to provide end-to-end encryption (E2EE) for user private data. We consider the following components as our trusted computing base (TCB): the hardware, the operating system (OS), and the framework that controls and mediates access to hardware. In the absence of physical input devices (e.g. mouse and keyboard) on mobile devices, we additionally trust the soft keyboard to not leak the keystrokes of a user. We rely on the TCB to correctly handle I/O for M-Aegis, and to provide proper isolation between M-Aegis and untrusted components.

Additionally, we also assume that all the components of M-Aegis, including L-7.5 that it creates, are trusted. The user is also considered trustworthy under our threat model in his intent. This means that he is trusted to turn on M-Aegis when he wants to protect the privacy of his data during his interaction with the PCS.

3.2.2 Out of Scope Threats

Our threat model does not consider the following types of attacks. First, M-Aegis only guarantees the confidentiality of a user's data, but not its availability. Therefore, attacks that deny access to data (denial-of-service) either at the server or the client are beyond the scope of this work. Second, any attacks against our TCB are orthogonal to this work. Such attacks include malicious hardware [52], attacks against the hardware [66], the OS [50], the platform [63] and privilege escalation attacks (e.g. unauthorized rooting of device). However, note that M-Aegis can be implemented on a design that anchors its trust on trusted hardware and hypervisor (e.g. Gyrus [48], Storage Capsules [29]) to minimize the attack surface against the TCB. Third, M-Aegis is designed to prevent any direct flow of information from an authorized user to untrusted entities. Hence, leakages through all side-channels [62] are beyond the scope of this work.

Since the user is assumed to be trustworthy under our threat model to use M-Aegis correctly, M-Aegis does not protect the user against social-engineering-based attacks. For example, phishing attacks to trick users into either turning off M-Aegis and/or entering sensitive information into unprotected UI components are beyond

the scope of our paper. Instead, M-Aegis deploys best-effort protection by coloring the UI components in L-7.5 differently from that of the default app UI.

The other limitations of M-Aegis, which are not security threats, are discussed in Sect. 6.2.

3.3 M-Aegis Architecture

M-Aegis is architected to fulfill all of the three design goals mentioned in Sect. 3.1. Providing strong isolation guarantees is first. To achieve this, M-Aegis is designed to execute in a separate process, though it resides in the same OS as the target client app (TCA). Besides memory isolation, the filesystem of M-Aegis is also shielded from other apps by OS app sandbox protection.

Should a greater degree of isolation be desirable, an underlying virtual-machine-based system can be adopted to provide even stronger security guarantees. However, we do not consider such design at this time as it is currently unsuitable for mobile platforms, and the adoption of such technology is beyond the scope of our paper. The main components that make up M-Aegis are as follows.

3.3.1 Layer 7.5 (L-7.5)

M-Aegis creates a novel and conceptual layer called Layer 7.5 (L-7.5) to interpose itself between the user and the TCA. This allows M-Aegis to implement true end-to-end encryption (E2EE) without exposing plaintext data to the TCA while maintaining the TCA's original functionalities and user experience, fulfilling the second design goal. L-7.5 is built by creating a transparent window that is always-on-top. This technique is advantageous in that it provides a natural way to handle user interaction, thus preserving user experience without the need to reverse engineer the logic of TCAs or the network protocols used by the TCAs to communicate with their respective cloud service backends, fulfilling the third design goal.

There are three cases of user interactions to handle. The first case considers interactions that do not involve data confidentiality (e.g. deleting or relabeling email). Such input do not require extra processing/transformation and can be directly delivered to the underlying TCA. Such click-through behavior is a natural property of transparent windows, and helps M-Aegis maintain the look and feel of the TCA.

The second case considers interactions that involve data confidentiality (e.g. entering messages or searching encrypted email). Such input requires extra processing (e.g. encryption and encoding operations). For such cases, M-Aegis places opaque GUIs that "mimic" the GUIs over the TCA, which are purposely painted in different colors for two reasons: 1) as a placeholder for user input so that it does not leak to the TCA, and 2) for user visual feedback. Mimic GUIs for the subject and content as seen in Fig. 3 are examples of this case. Since L-7.5 is always on top, this provides the guarantee that user input always goes to a mimic GUI instead of the TCA.

The third case considers interactions with control GUIs (e.g. send buttons). Such input requires user action to be "buffered" while the input from the second case is being processed before being relayed to the actual control GUI of the TCA. For such cases, M-Aegis creates semi-transparent mimic GUIs that register themselves to absorb/handle user clicks/taps. Again, these mimic GUIs are painted with a different color to provide a visual cue to a user. Examples of these include the purple search button in the left figure in Fig. 2 and the purple send button in Fig. 3. Note that our concept of intercepting user input is similar to that of ACG's [57] in capturing user intent, but our application of user intent differs.

3.3.2 UIA Manager (UIAM)

To be fully functional, there are certain capabilities that M-Aegis requires but are not available to normal apps. First, although M-Aegis is confined within the OS' app sandbox, it must be able to determine with which TCA the user is currently interacting. This allows M-Aegis to invoke specific logic to handle the TCA, and helps M-Aegis clean up the screen when the TCA is terminated. Second, M-Aegis requires information about the GUI layout for the TCA it is currently handling. This allows M-Aegis to properly render mimic GUIs on L-7.5 to intercept user I/O. Third, although isolated from the TCA, M-Aegis must be able to communicate with the TCA to maintain functionality and ensure user experience is not disrupted. For example, M-Aegis must be able to relay user clicks to the TCA, eventually send encrypted data to the TCA, and click on TCA's button on behalf of the user. For output on screen, it must be able to capture ciphertext so that it can decrypt it and then render it on L-7.5.

M-Aegis extracts certain features from the underlying OS's accessibility framework, which are exposed to developers in the form of User Interface Accessibility/Automation (UIA) library. Using UIA, M-Aegis is not only able to know which TCA is currently executing, but it can also query the GUI tree of the TCA to get detailed information about how the page is laid out (e.g. location, size, type, and resource-id of the GUI components). More importantly, it is able to obtain information about the content of these GUI items.

Exploiting UIA is advantageous to our design as compared to other methods of information capture from the GUI, e.g. OCR. Besides having perfect content accuracy, our technique is not limited by screen size. For example,

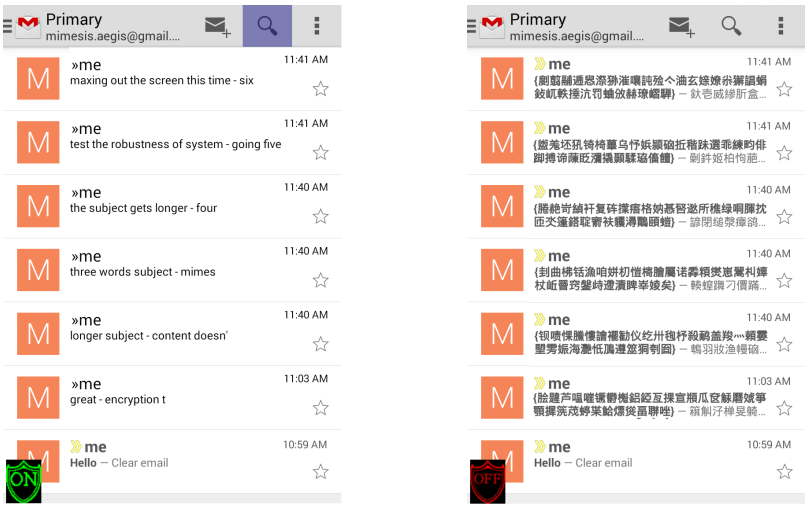


Figure 2: The figure on the left illustrates how a user perceives the Gmail preview page when M-Aegis is turned on. The figure on the right illustrates the same scenario but with M-Aegis turned off. Note that the search button is painted with a different color when M-Aegis is turned on.

even though the screen size may prevent full text to be displayed, M-Aegis is still able to capture text in its *entirety* through the UIA libraries, allowing us to comfortably apply decryption to ciphertext.

We thus utilize all these capabilities and advantages to build a crucial component of M-Aegis called the UIA manager (UIAM).

3.3.3 Per-TCA Logic

M-Aegis can be extended to support many TCAs. For each TCA of interest, we build per-TCA logic as an extension module. The per-TCA logic is responsible for rendering the specific mimic GUIs according to information it queries from the UIAM. Therefore, per-TCA logic is responsible for handling direct user input. Specifically, it decides whether the user input will be directly passed to the TCA or be encrypted and encoded before doing so. This ensures that the TCA never obtains plaintext data while user interaction is in plaintext mode. Per-TCA logic also intercepts button clicks so that it can then instruct UIAM to emulate the user's action on the button in the underlying TCA. Per-TCA logic also decides which encryption and encoding scheme to use according to the type of TCA it is handling. For example, encryption and encoding schemes for handling email apps would differ from that of messenger apps.

3.3.4 Cryptographic Module

M-Aegis' cryptographic module is responsible for providing encryption/decryption and cryptographic hash capabilities to support our searchable encryption scheme (described in detail later) to the per-TCA logic the so that M-Aegis can transform/obfuscate messages through E2EE operations. Besides standard cryptographic primitives, this module also includes a searchable encryption scheme to support search over encrypted email that works *without server modification*. Since the discussion of any encryption scheme is not complete without encryption keys, key manager is also a part of this module.

Key Manager. M-Aegis has a key manager per TCA that manages key policies that can be specific to each TCA according to user preference. The key manager supports a range of schemes, including simple password-based key derivation functions (of which we assume the password to be shared out of band) to derive symmetric keys, which we currently implement as default, to more sophisticated PKI-based scheme for users who prefer stronger security guarantees and do not mind the additional key set-up and exchange overheads. However, the discussion about the best key management/distribution policy is beyond the scope of this paper.

Searchable Encryption Scheme (EDESE). There are numerous encryption schemes that support keyword search [45, 61, 44, 31, 33, 28, 51]. These schemes exhibit different tradeoffs between security, functionality and efficiency, but *all* of them require modifications on the server side. Schemes that make use of inverted index [33] are not suitable, as updates to inverted index cannot be practically deployed in our scenario.

Since we cannot assume server cooperation (consistent with our threat model in Sect. 3.2), we designed a new searchable encryption scheme called easily-deployable efficiently-searchable symmetric encryption scheme (EDESE). EDESE is an adaptation of a scheme

proposed by Bellare et al. [25], with modifications similar to that of Goh's scheme [44] that is retrofittable to a non-modifying server scenario.

We incorporated EDESE for email applications with the following construct. The idea for the construction is simple: we encrypt the document with a standard encryption scheme and append HMACs of unique keywords in the document. We discuss the specific instantiations of encryption and HMAC schemes that we use in Sect. 4.1. To prevent leaking the number of unique keywords we add as many "dummy" keywords as needed. We present this construction in detail in the full version of our paper [54].

In order to achieve higher storage and search efficiency, we utilized a Bloom filter (BF) to represent the EDESE-index. Basically, a BF is a data structure that allows for efficient set-inclusion tests. However, such set-inclusion tests based on BFs are currently not supported by existing email providers, which only support string-based searches. Therefore, we devised a solution that encodes the positions of on-bits in a BF as Unicode strings (refer to Sect. 4.4 for details).

Since the underlying data structure that is used to support EDESE is a BF, search operations are susceptible to false positives matches. However, this does not pose a real problem to users, because the false positive rate is extremely low and is completely adjustable. Our current implementation follows these parameters: the length of keyword (in bits) is estimated to be $k = 128$, the size of the BF array is $B = 2^{24}$, the maximum number of unique keywords used in any email thread is estimated to be $d = 10^6$, the number of bits set to 1 for one keyword is $r = 10$. Plugging in these values into the formula for false positive calculation [44], i.e. $(1 - e^{-rd/B})^r$, we cap the probability of a false positive δ to 0.0003.

We formally assess the security guarantees that our construction provides. In the full version of our paper [54], we propose a security definition for EDESE schemes and discuss why the existing notions are not suitable. Our definition considers an attacker who can obtain examples of encrypted documents of its choice and the results of queries of keywords of its choice. Given such an adversary, an EDESE scheme secure under our definition should hide all partial information about the messages except for the message length and the number of common keywords between any set of messages. Leaking the latter is unavoidable given that for the search function to be transparent to encryption, the output of a query has to be a part a ciphertext. But everything else, e.g., the number of unique keywords in a message, positions of the keywords, is hidden.

Given the security definition in our full paper [54], we prove that our construction satisfies it under the standard notions of security for encryption and HMACs.

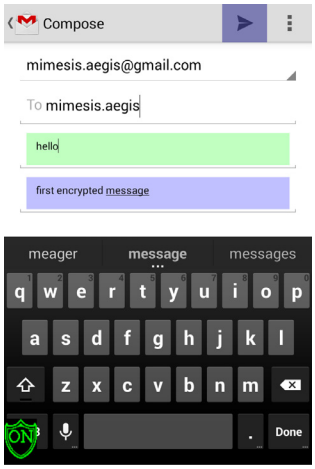


Figure 3: User still interacts with Gmail app to compose email, with M-Aegis' mimic GUIs painted with different colors on L-7.5.

3.4 User Workflow

To better illustrate how the different components in M-Aegis fit together, we describe an example workflow of a user composing and sending an email using the stock Gmail app on Android using M-Aegis:

1) When the user launches the Gmail app, the UIAM notifies the correct per-TCA logic of the event. The per-TCA logic will then initialize itself to handle the Gmail workflow.

2) As soon as Gmail is launched, the per-TCA logic will try to detect the state of Gmail app (e.g. preview, reading, or composing email). This allows M-Aegis to properly create mimic GUIs on L-7.5 to handle user interaction. For example, when a user is on the compose page, the per-TCA logic will mimic the GUIs of the subject and content fields (as seen in Fig. 3). The user then interacts directly with these mimic GUIs in plaintext mode without extra effort. Thus, the workflow is not affected. Note that essential but subtle features like spell check and autocorrect are still preserved, as they are innate features of the mobile device's soft keyboard. Additionally, the "send" button is also mimicked to capture user intent.

3) As the user finishes composing his email, he clicks on the mimicked "send" button on L-7.5. Since L-7.5 receives the user input and not the underlying Gmail app, the per-TCA logic is able to capture this event and proceed to process the subject and the content.

4) The per-TCA logic selects the appropriate encryption key to be used based on the recipient list and the predetermined key policy for Gmail. If a key cannot be found for this conversation, M-Aegis prompts the user (see Fig. 4) for a password to derive a new key. After ob-

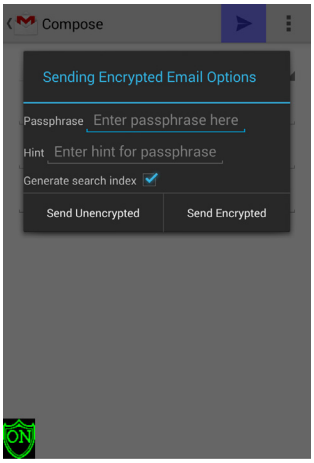


Figure 4: Password prompt when user sends encrypted mail for a new conversation.

taining the associated key for this conversation, M-Aegis will then encrypt these inputs and encode it back to text such that Gmail can consume it.

5) The per-TCA logic then requests the UIAM to fill in the corresponding GUIs on Gmail with the transformed text. After they are filled, the UIAM is instructed to click the actual "send" button on behalf of the user. This provides a transparent experience to the user.

From this workflow, it should therefore be evident that from the user's perspective, the workflow of using Gmail remains the same, because of the mimicking properties of M-Aegis.

4 Implementation and Deployment

In this section, we discuss important details of our prototype implementations. We implemented a prototype of M-Aegis using Java on Android, as an accessibility service. This is done by creating a class that extends the `AccessibilityService` class and requesting the `BIND_ACCESSIBILITY_SERVICE` permission in the manifest. This allows us to interface with the UIA library, building our UIAM. We discuss this in further detail in Sect. 4.2.

We then deployed our prototype on two Android phones from separate manufacturers, i.e. Samsung Galaxy Nexus, and LG Nexus 4, targeting several versions of Android, from Android 4.2.2 (API level 17) to Android 4.4.2 (API level 19). The deployment was done on stock devices and OS, i.e. *without* modifying the OS, Android framework, or rooting. Only simple app installation was performed. This demonstrates the ease of deployment and distribution of our solution. We have also implemented an M-Aegis prototype on Windows 7 to demonstrate interoperability and generality of approach, but we do not discuss the details here, as it is not the focus of this paper.

As an interface to the user, we create a button that is always on top even if other apps are launched. This allows us to create a non-bypassable direct channel of communication with the user besides providing visual cue of whether M-Aegis is turned on or off.

For app support, we use Gmail as an example of an email app and WhatsApp as an example of a messenger app. We argue that it is easy to extend the support to other apps within these classes.

We first describe the cryptographic schemes that we deployed in our prototype, then we explain how we build our UIAM and create L-7.5 on Android, and finally discuss the per-TCA logic required to support both classes of apps.

4.1 Cryptographic Schemes

For all the encryption/decryption operations, we use AES-GCM-256. For a password-based key generation algorithm, we utilized PBKDF2 with SHA-1 as the keyed-hash message authentication code (HMAC). We also utilized HMAC-SHA-256 as our HMAC to generate tags for email messages (Sect. 4.4.1). These functionalities are available in Java's `javax.crypto` and `java.security` packages.

For the sake of usability, we implemented a password-based scheme as the default, and we assume one password for each group of message recipients. We rely on the users to communicate the password to the receiving parties using out of band channel (e.g. in person or phone calls). For messaging apps, we implemented an authenticated Diffie-Hellman key exchange protocol to negotiate session keys for WhatsApp conversations. A PGP key is automatically generated for a user during installation based on the hashed phone number, and is deposited to publicly accessible repositories on the user's behalf (e.g. MIT PGP Key Server [10]). Further discussion about verifying the authenticity of public keys retrieved from such servers is omitted from this paper. Since all session and private keys are stored locally for user convenience, we make sure that they are never saved to disk in plaintext. They are additionally encrypted with a key derived from a master password that is provided by the user during installation.

4.2 UIAM

As mentioned earlier, UIAM is implemented using UIA libraries. On Android, events that signify something new being displayed on the screen can be detected by monitoring following the events: `WINDOW_CONTENT_CHANGED`, `WINDOW_STATE_CHANGED`,

and VIEW_SCROLLED. Upon receiving these events, per-TCA logic is informed. The UIA library presents a data structure in the form of a tree with nodes representing UI components with the root being the top window. This allows the UIAM to locate all UI components on the screen.

Additionally, Android's UIA framework also provides the ability to query for UI nodes by providing a resource ID. For instance, the node that represents Gmail's search button can be found by querying for com.google.android.gm:id/search. More importantly, there is no need to guess the names of these resource IDs. Rather, a tool called UI Automator Viewer [17] (see Sect. 4.4), which comes with the default Android SDK. Once the node of interest is found, all the other information about the GUI represented by the node can be found. This includes the exact location and size of text boxes and buttons on the screen.

M-Aegis is able to programmatically interact with various GUIs of a TCA using the function performAction(). This allows it t to click on a TCA's button on the user's behalf after it has processed the user input.

4.3 Layer 7.5

We implemented Layer 7.5 on Android as specific types of system windows, which are *always-on-top* of all other running apps. Android allows the creation of various types of system windows. We focus on two, TYPE_SYSTEM_OVERLAY and TYPE_SYSTEM_ERROR; the first is for display only and allows all tap/keyboard events to go to underlying apps. In contrast, the second type allows for user interaction. Android allows the use of any View objects for either type of window, and we use this to create our mimic GUIs, and set their size and location. We deliberately create our mimic GUIs in different colors as a subtle visual cue to the users that they are interacting with M-Aegis, without distracting them from their original workflow.

4.4 Per-TCA Logic

From our experience developing per-TCA logic, the general procedure for development is as follows:

1) Understand what the app does. This allows us to identify which GUIs need to be mimicked for intercepting user I/O. For text-based TCAs, this is a trivial step because the core functionalities that M-Aegis needs to handle are limited and thus easy to identify, e.g. buffering user's typed texts and sending them to the intended recipient.

2) Using UI Automator Viewer [17], examine the UIA tree for the relevant GUIs of a TCA and identify signatures (GUI resource IDs) for each TCA state. UI Automator Viewer allows inspection of the UIA tree through a graphical interface (as seen in Fig. 5), which reduces development time. We rely on UI components that are unique to certain states (e.g. the "send" button signifies that we are in the compose state).

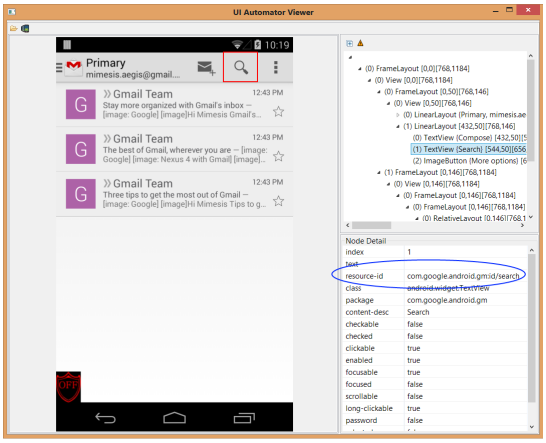


Figure 5: The UI Automator Viewer presents an easy to use interface to examine the UIA tree and determine the resource ID (blue ellipse) associated with a GUI of interest (red rectangle).

3) For each relevant GUI, we need to devise algorithms to extract either the location and content of ciphertext (for decryption and display), or the type, size, and location of GUIs we need to mimic (e.g. the subject and content boxes in the Gmail compose UI). Again, this is done through UI Automator Viewer. For example, for the Gmail preview state, we query the UIA for nodes with ID com.google.android.gm:/id/conversation_list to identify all the UIA nodes corresponding to the preview item of each individual email, and from those we can extract all ciphertext on the preview window through the UIA).

4) Create event handlers for controls we mimic on L-7.5. For the Gmail compose state, we need to listen for click/touch events for the L-7.5 "send" button and carry out the process described in Sect. 3.3.3 to encrypt the email and send the ciphertext to the underlying TCA.

5) Identify ways that each relevant state can be updated. Updates can be handled via the following method: clear L-7.5, extract all necessary information from the new state, and render again. This is equivalent to redrawing all GUIs on L-7.5 based on the detected state.

There are two details worth considering when developing per-TCA logic. First, careful consideration must be given about the type of input data fed to TCAs. Since most TCAs only accept input data in specific formats, e.g. text, they do not support the input of random byte sequences as valid data. Therefore, encrypted data must

be encoded into text format before feeding it as input to a TCA. Conventionally, base64 encoding is used for this purpose. However, base64 encoding consumes too much on-screen real estate. To overcome this, we encoded the binary encrypted data into Chinese Japanese Korean (CJK) Unicode characters, which have efficient on-screen real estate consumption. To map the binary data into the CJK plane, we process the encrypted data at the byte granularity (2^8). For each byte, its value is added to the base of the CJK Unicode representation, i.e. 0x4E00. For example, byte 0x00 will be encoded as '一', and byte 0x01 will be represented as '丁'.

Second, M-Aegis can only function correctly if it can differentiate between ordinary messages and encrypted messages. We introduce markers into the encrypted data after encoding; in particular, we wrap the subject and content of a message using a pair of curly braces (i.e. {, }).

Next, we describe implementation details that are specific to these classes of apps. We begin by introducing the format of message we created for each class. Then we discuss other caveats (if any) that are involved in the implementation.

4.4.1 Email Apps

We implemented support for Gmail on our prototype as a representative app of this category. We create two custom formats to communicate the necessary metadata to support M-Aegis' functionalities.
Subject: $\{Encode(ID_{Key}||IV||Encrypt(Subject))\}$
Content: $\{Encode(Encrypt(Content)||Tags)\}$

A particular challenge we faced in supporting decryption during the Gmail preview state is that only the beginning parts of both the title and the subject of each message are available to us. Also, the exact email addresses of the sender and recipients are not always available, as some are displayed as aliases, and some are hidden due to lack of space. The lack of such information makes it impossible to automatically decrypt the message even if the corresponding encryption key actually exists on the system.

To solve these problems, when we encrypt a message, we include a key-ID (ID_{Key}) to the subject field (as seen in the format description above). Note that since the key-ID is not a secret, it need not be encrypted. This way, we will have all the information we need to correctly decrypt the subtext displayed on the Gmail preview.

The *Tags* field is a collection of HMAC digests that are computed using the conversation key and keywords that exist in a particular email. It is then encoded and appended as part of the content that Gmail receives to facilitate encrypted search without requiring modification to Gmail's servers.

4.4.2 Messenger Apps

We implemented support for WhatsApp on our prototype as a representative app of this category. The format we created for this class of apps is simple, as seen below:
Message: $\{Encode(IV||Encrypt(Message))\}$

We did not experience additional challenges when supporting WhatsApp.

5 Evaluations

In this section, we report the results of experiments to determine the correctness of our prototype implementation, measure the overheads of M-Aegis, and user acceptability of our approach.

5.1 Correctness of Implementation

We manually verified M-Aegis's correctness by navigating through different states of the app and checking if M-Aegis creates L-7.5 correctly. We manually verified that the encryption and decryption operations of M-Aegis work correctly. We ensured that plaintext is properly received at the recipient's end when the correct password is supplied. We manually verified the correctness of our searchable encryption scheme by introducing specific search keywords. We performed search using M-Aegis and found no false negatives in the search result.

5.2 Performance on Android

The overhead that M-Aegis introduced to a user's workflow can be broken down into two factors: the additional computational costs incurred during encryption and decryption of data, and the additional I/O operations when redrawing L-7.5. We measure overhead by measuring the overall latency presented to the user in various use cases. We found that M-Aegis imposes negligible latency to the user.

All test cases were performed on a *stock* Android phone (LG Nexus 4), with the following specifications: Quad-core 1.5 GHz Snapdragon S4 Pro CPU, equipped with 2.0 GB RAM, running Android Kit Kat (4.4.2, API level 19). Unless otherwise stated, each experiment is repeated 10 times and the averaged result is reported.

For our evaluation, we only performed experiments for the setup of the Gmail app because Gmail is representative of a more sophisticated TCA, and thus indicates worst-case performance for M-Aegis. Messenger apps incur fewer overheads given their simpler TCA logic.

5.2.1 Previewing Encrypted Email

There are additional costs involved in previewing encrypted emails on the main page of Gmail. The costs are

broken down into times taken to 1) traverse the UIA tree to identify preview nodes, 2) capture ciphertext from the UIA node, 3) obtain the associated encryption key from the key manager, 4) decrypting ciphertext, and 5) rendering plaintext on L-7.5. We measure these operations as a single entity by running a macro benchmark.

For our experiment, we ensured that the preview page consists of encrypted emails (a total of 6 can fit on-screen) to demonstrate worst-case performance. We measured the time taken to perform all operations. We found, on average, it takes an additional 76 ms to render plaintext on L-7.5. Note that this latency is well within expected response time (50 - 150 ms), beyond which a user would notice the slowdown effect [60].

5.2.2 Composing and Sending Encrypted Email

We measured the extra time taken for a typical email to be encrypted and for our searchable encryption index to be built. We used the Enron Email Dataset [32] as a representation of typical emails. We randomly picked 10 emails. The average number of words in an email is 331, of which 153 are unique. The shortest sampled email contained 36 words, of which 35 are unique. The longest sampled email contains 953 words, of which 362 are unique.

With the longest sampled email, M-Aegis took 205 ms in total to both encrypt and build the search index. Note that this includes the network latency a user will perceive while sending an email, regardless of their use of M-Aegis.

5.2.3 Searching on Encrypted Emails

A user usually inputs one to three keywords per search operation. The latency experienced when performing search is negligible. This is because the transformation of the actual keyword into indexes requires only the forward computation of one HMAC, which is nearly instantaneous.

5.3 User Acceptability Study

This section describes the user study we performed to validate our hypothesis of user acceptability of M-Aegis. Users were sampled from a population of college students. They must be able to proficiently operate smart phones and have had previous experience using the Gmail app. Each experiment was conducted with two identical smart phones, i.e. Nexus 4, both running Android 4.3, installed with the stock Gmail app (v. 4.6). One of the devices had M-Aegis installed.

The set up of the experiment is as follows. We asked the user to perform a list of tasks: previewing, reading, composing, sending, and searching through email on a device that is not equipped with M-Aegis. Participants were asked to pay attention to the overall experience of performing such tasks using the Gmail app. This served as the control experiment.

Participants were then told to repeat the same set of tasks on another device that was equipped with M-Aegis. This was done with the intention that they were able to mentally compare the difference in user experience when interacting with the two devices.

We queried the participants if they found any difference in the preview page, reading, sending, and searching email, and if they felt that their overall experience using the Gmail app on the second device was significantly different.

We debriefed the participants about the experiment process and explained the goal of M-Aegis. We asked them whether they would use M-Aegis to protect the privacy of their data. The results we collected and report here are from 15 participants.

We found that no participants noticed major differences between the two experiences using the Gmail app. One participant noticed a minor difference in the email preview interface, i.e. L-7.5 did not catch up smoothly when scrolled. A different participant noticed a minor difference in the process of reading email, i.e. L-7.5 lag before covering ciphertext with mimic GUIs. There were only two participants that found the process of sending email differed from the original. When asked for details, they indicated that the cursor when composing email was not working properly. After further investigation, we determined this was a bug in Android's GUI framework rather than a fundamental flaw in M-Aegis's design.

Despite the perceived minor differences when performing particular tasks, all participants indicated that they would use M-Aegis to protect the privacy of their data after understanding what M-Aegis is. This implies that they believe that the overall disturbance to the user experience is not large enough to impede adoption.

Since we recruited 15 users for this study, the accuracy/quality of our conclusion from this study lies between 80% and 95% (between 10 and 20 users) according to findings in [42]. We intend to continue our user study to further validate our acceptability hypothesis and to continuously improve our prototype based on received feedback.

6 Discussions

6.1 Generality and Scalability

We believe that our M-Aegis architecture presents a general solution that protects user data confidentiality, which is scalable in the following aspects:

Across Multiple Cloud Services. There are two main classes of apps that provide communication services, email and messenger apps. By providing functionality for apps in these two categories, we argue that M-Aegis can satisfy a large portion of user mobile security needs. The different components of M-Aegis incur a one-time development cost. We argue that it is easy to scale across multiple cloud services, because per-TCA logic that needs to be written is minimal per new TCA. This should be evident through the five general steps highlighted in Sect. 4.4. In addition, the logic we developed for the first TCA (Gmail) serves as a template/example to implement support for other apps.

Across App Updates. Since the robustness of the UIAM construct (Sect. 4.2) gives M-Aegis the ability to track all TCA GUIs regardless of TCA state, M-Aegis is able to easily survive app updates. Our Gmail app support has survived two updates without requiring major efforts to adapt.

Resource ID names can change across updates. For example, when upgrading to Gmail app version 4.7.2, the resource ID name that identifies a sender's account name changed. Using UI Automator Viewer, we quickly discovered and modified the mapping in our TCA logic. Note that only the mapping was changed; the logic for the TCA does not need to be modified. This is because the core functionality of the updated GUI did not change (i.e., the GUI associated with a sender's account remained a text input box).

6.2 Limitations

As mentioned earlier, M-Aegis is not designed to protect users against social-engineering-based attacks. Adversaries can trick users into entering sensitive information to the TCA while M-Aegis is turned off. Our solution is best effort by providing distinguishing visual cues to the user when M-Aegis is turned on and its L-7.5 is active. For example the mimic GUIs that M-Aegis creates a different color. Users can toggle M-Aegis' button on or off to see the difference (see Fig. 2). Note that M-Aegis's main button is always on top and cannot be drawn over by other apps. However, we do not claim that this fully mitigates the problem.

One of the constraints we faced while retrofitting a security solution to existing TCAs (not limited to mobile environments) is that data must usually be of the right format (e.g. strictly text, image, audio, or video). For example, Gmail accepts only text (Unicode-compatible) for an email subject, but Dropbox accepts any type of files, including random blobs of bytes. Currently, other than text format, we do not yet support other types of user data (e.g. image, audio and video). However, this is *not* a fundamental design limitation of our system. Rather, it is because of the unavailability of transformation functions (encryption and encoding schemes) that works for the these media types.

Unlike text, the transformation/obfuscation functions in M-Aegis for other type of data may also need to survive other process steps, such as compression. It is normal for TCAs to perform compression on multimedia to conserve bandwidth and/or storage. For example, Facebook is known to compress/downsample the image uploads.

The confidentiality guarantee that we provide excludes risks at the end points themselves. For example, a poor random number generator can potentially weaken the cryptographic schemes M-Aegis applies. It is currently unclear how our text transformations will affect a server's effectiveness in performing spam filtering.

Our system currently does not tolerate typographical error during search. However, we would like to point out that this is an unlikely scenario, given that soft keyboards on mobile devices utilize spell check and autocorrect features. Again, this is not a flaw with our architecture; rather, it is because of the unavailability of encryption schemes that tolerate typographical error search without requiring server modification.

7 Conclusions

In this paper we presented *Mimesis Aegis (M-Aegis)*, a new approach to protect private user data in public cloud services. M-Aegis provides strong isolation and preserves user experience through the creation of a novel conceptual layer called *Layer 7.5 (L-7.5)*, which acts as a proxy between an app (Layer 7) and a user (Layer 8). This approach allows M-Aegis to implement true end-to-end encryption of user data while achieving three goals: 1) plaintext data is never visible to a client app, any intermediary entities, or the cloud provider; 2) the original user experience with the client app is preserved completely, from workflow to GUI look-and-feel; and 3) the architecture and technique are general to a large number of apps and resilient to app updates. We implemented a prototype of M-Aegis on Android that can support a number of popular cloud services (e.g. Gmail, Google Hangout, Facebook, WhatsApp, and Viber). Our user study shows that our system preserves both the workflow and the GUI look-and-feel of the protected applications, and our performance evaluations show that users experienced minimal overhead in utilizing M-Aegis on Android.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments. We also thank the vari-

ous members of our operations staff who provided proofreading of this paper. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, CNS-1149051, and CNS-1318511, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] Accessibility. `http://developer.android.com/guide/topics/ui/accessibility/index.html`.

[2] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.

[3] Cryptocat. https://crypto.cat.

[4] Engineering Security Solutions at Layer 8 and Above. https://blogs.rsa.com/engineering-security-solutions-at-layer-8-and-above/, December.

[5] Gibberbot for Android devices. https://securityinabox.org/en/Gibberbot_main.

[6] Google Accessibility. `https://www.google.com/accessibility/policy/`.

[7] Google Chrome Mobile FAQ. https://developers.google.com/chrome/mobile/docs/faq.

[8] International technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. http://www.ecma-international.org/activities/Communications/TG11/s020269e.pdf.

[9] Microsoft Accessibility. `http://www.microsoft.com/enable/microsoft/section508.aspx`.

[10] MIT PGP Public Key Server. http://pgp.mit.edu/.

[11] New privacy fears as facebook begins selling personal access to companies to boost ailing profits. http://www.dailymail.co.uk/news/article-2212178/New-privacy-row-Facebook-begins-selling-access-users-boost-ailing-profits.html.

[12] Secure texts for Android. https://whispersystems.org.

[13] Sniffer tool displays other people's WhatsApp messages. http://www.h-online.com/security/news/item/Sniffer-tool-displays-other-people-s-WhatsApp-messages-1574382.html.

[14] Snowden: Leak of NSA spy programs "marks my end". http://www.cbsnews.com/8301-201_162-57588462/snowden-leak-of-nsa-spy-programs-marks-my-end/.

[15] Symantec desktop email encryption end-to-end email encryption software for laptops and desktops. http://www.symantec.com/desktop-email-encryption.

[16] Ten Mistakes That Can Ruin Customers' Mobile App Experience. http://www.itbusinessedge.com/slideshows/show.aspx?c=96038.

[17] UI Testing — Android Developers. `http://developer.android.com/tools/testing/testing_ui.html`.

[18] Whatsapp is broken, really broken. http://fileperms.org/whatsapp-is-broken-really-broken/.

[19] Layer 8 Linux Security: OPSEC for Linux Common Users, Developers and Systems Administrators. http://www.linuxgazette.net/164/kachold.html, July 2009.

[20] Accessibility. `https://www.apple.com/accessibility/resources/`, February 2014.

[21] 107TH CONGRESS. Uniting and strengthening america by providing appropriate tools required to intercept and obstruct terrorism (usa patriot act) act of 2001. *Public Law 107-56* (2001).

[22] ACQUISTI, A., AND GROSS, R. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *Privacy enhancing technologies* (2006), Springer, pp. 36–58.

[23] BADEN, R., BENDER, A., SPRING, N., BHATTACHARJEE, B., AND STARIN, D. Persona: an online social network with user-defined privacy. In *ACM SIGCOMM Computer Communication Review* (2009), vol. 39, ACM, pp. 135–146.

[24] BEATO, F., KOHLWEISS, M., AND WOUTERS, K. Scramble! your social network data. In *Privacy Enhancing Technologies* (2011), Springer, pp. 211–225.

[25] BELLARE, M., BOLDYREVA, A., AND O'NEILL, A. Deterministic and efficiently searchable encryption. In *CRYPTO* (2007), A. Menezes, Ed., vol. 4622 of *Lecture Notes in Computer Science*, Springer, pp. 535–552.

[26] BERTHOME, P., FECHEROLLE, T., GUILLOTEAU, N., AND LALANDE, J.-F. Repackaging android applications for auditing access to private data. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on* (2012), IEEE, pp. 388–396.

[27] BÖHMER, M., HECHT, B., SCHÖNING, J., KRÜGER, A., AND BAUER, G. Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In *Proceedings of the 13th international conference on Human computer interaction with mobile devices and services* (2011), ACM, pp. 47–56.

[28] BONEH, D., CRESCENZO, G. D., OSTROVSKY, R., AND PERSIANO, G. Public key encryption with keyword search. In *EUROCRYPT* (2004), C. Cachin and J. Camenisch, Eds., vol. 3027 of *Lecture Notes in Computer Science*, Springer, pp. 506–522.

[29] BORDERS, K., VANDER WEELE, E., LAU, B., AND PRAKASH, A. Protecting confidential data on personal computers with storage capsules. *Ann Arbor 1001* (2009), 48109.

[30] BORISOV, N., GOLDBERG, I., AND BREWER, E. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society* (2004), ACM, pp. 77–84.

[31] CHANG, Y.-C., AND MITZENMACHER, M. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, J. Ioannidis, A. Keromytis, and M. Yung, Eds., vol. 3531 of *Lecture Notes in Computer Science*. Springer, 2005, pp. 442–455.

[32] COHEN, W. W. Enron email dataset. http://www.cs.cmu.edu/ enron, August 2009.

[33] CURTMOLA, R., GARAY, J. A., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security* (2006), A. Juels, R. N. Wright, and S. D. C. di Vimercati, Eds., ACM, pp. 79–88.

[34] DELTCHEVA, R. Apple, AT&T data leak protection issues latest in cloud failures. http://www.messagingarchitects.com/resources/security-compliance-news/email-security/apple-att-data-leak-protection-issues-latest-in-cloud-failures19836720.html, June 2010.

[35] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. Tech. rep., DTIC Document, 2004.

[36] DONG, X., CHEN, Z., SIADATI, H., TOPLE, S., SAXENA, P., AND LIANG, Z. Protecting sensitive web content from client-side vulnerabilities with cryptons. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1311–1324.

[37] ELKINS, M. Mime security with pretty good privacy (pgp).

[38] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010), vol. 10, pp. 1–6.

[39] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 50–61.

[40] FAHL, S., HARBACH, M., MUDERS, T., AND SMITH, M. Trust-split: usable confidentiality for social network messaging. In *Proceedings of the 23rd ACM conference on Hypertext and social media* (2012), ACM, pp. 145–154.

[41] FARB, M., LIN, Y.-H., KIM, T. H.-J., MCCUNE, J., AND PERRIG, A. Safeslinger: easy-to-use and secure public-key exchange. In *Proceedings of the 19th annual international conference on Mobile computing & networking* (2013), ACM, pp. 417–428.

[42] FAULKNER, L. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers 35*, 3 (2003), 379–383.

[43] FELDMAN, A. J., BLANKSTEIN, A., FREEDMAN, M. J., AND FELTEN, E. W. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX conference on Security symposium, Security* (2012), vol. 12.

[44] GOH, E.-J. Secure indexes. *IACR Cryptology ePrint Archive* (2003).

[45] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (1996), 431–473.

[46] GUHA, S., TANG, K., AND FRANCIS, P. Noyb: Privacy in online social networks. In *Proceedings of the first workshop on Online social networks* (2008), ACM, pp. 49–54.

[47] HENRY, S. Largest hacking, data breach prosecution in U.S. history launches with five arrests. http://www.mercurynews.com/business/ci23730361/largest-hacking-data-breach-prosecution-u-s-history, July 2013.

[48] JANG, Y., CHUNG, S. P., PAYNE, B. D., AND LEE, W. Gyrus: A framework for user-intent monitoring of text-based networked applications. In *NDSS* (2014).

[49] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (2012), ACM, pp. 3–14.

[50] JIANG, X. Gingermaster: First android malware utilizing a root exploit on android 2.3 (gingerbread). `http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/`.

[51] KAMARA, S., PAPAMANTHOU, C., AND ROEDER, T. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 965–976.

[52] KING, S. T., TUCEK, J., COZZIE, A., GRIER, C., JIANG, W., AND ZHOU, Y. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, USA, 2008), LEET'08, USENIX Association, pp. 5:1–5:8.

[53] KONTAXIS, G., POLYCHRONAKIS, M., KEROMYTIS, A. D., AND MARKATOS, E. P. Privacy-preserving social plugins. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 30–30.

[54] LAU, B., CHUNG, S., SONG, C., JANG, Y., LEE, W., AND BOLDYREVA, A. Mimesis Aegis: A Mimicry Privacy Shield. http://hdl.handle.net/1853/52026.

[55] LUCAS, M. M., AND BORISOV, N. Flybynight: mitigating the privacy risks of social networking. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society* (2008), ACM, pp. 1–8.

[56] PEEK, D., AND FLINN, J. Trapperkeeper: the case for using virtualization to add type awareness to file systems. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems* (2010), USENIX Association, pp. 8–8.

[57] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 224–238.

[58] SHAH, K. Common Mobile App Design Mistakes to Take Care. http://www.enterprisecioforum.com/en/blogs/kaushalshah/common-mobile-app-design-mistakes-take-c.

[59] SHENG, S., BRODERICK, L., HYLAND, J., AND KORANDA, C. Why johnny still can't encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security* (2006).

[60] SHNEIDERMAN, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, fourth ed. Addison-Wesley, 2005.

[61] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy* (2000), pp. 44–55.

[62] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of des implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems-CHES 2003*. Springer, 2003, pp. 62–76.

[63] US-CERT/NIST. Cve-2013-4787. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4787`.

[64] WHITTAKER, S., MATTHEWS, T., CERRUTI, J., BADENES, H., AND TANG, J. Am i wasting my time organizing email?: a study of email refinding. In *PART 5——Proceedings of the 2011 annual conference on Human factors in computing systems* (2011), ACM, pp. 3449–3458.

[65] WHITTEN, A., AND TYGAR, J. D. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium* (1999), vol. 99, McGraw-Hill.

[66] WOJTCZUK, R., AND TERESHKIN, A. Attacking intel® bios. *Invisible Things Lab* (2010).

[67] WU, C., ZHOU, Y., PATEL, K., LIANG, Z., AND JIANG, X. Airbag: Boosting smartphone resistance to malware infection. In *NDSS* (2014).

[68] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 27–27.

XRay: Enhancing the Web's Transparency with Differential Correlation

Mathias Lécuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios,
Riley Spahn, Augustin Chaintreau, *and* Roxana Geambasu
Columbia University

Abstract

Today's Web services – such as Google, Amazon, and Facebook – leverage user data for varied purposes, including personalizing recommendations, targeting advertisements, and adjusting prices. At present, users have little insight into how their data is being used. Hence, they cannot make informed choices about the services they choose.

To increase transparency, we developed *XRay*, the first fine-grained, robust, and scalable personal data tracking system for the Web. XRay predicts which data in an arbitrary Web account (such as emails, searches, or viewed products) is being used to target which outputs (such as ads, recommended products, or prices). XRay's core functions are service agnostic and easy to instantiate for new services, and they can track data within and across services. To make predictions independent of the audited service, XRay relies on the following insight: by comparing outputs from different accounts with similar, but not identical, subsets of data, one can pinpoint targeting through correlation. We show both theoretically, and through experiments on Gmail, Amazon, and YouTube, that XRay achieves high precision and recall by correlating data from a surprisingly small number of extra accounts.

1 Introduction

We live in a "big data" world. Staggering amounts of personal data – our as locations, search histories, emails, posts, and photos – are constantly collected and analyzed by Google, Amazon, Facebook, and a myriad of other Web services. This presents rich opportunities for marshaling big data to improve daily life and social well-being. For example, personal data improves the usability of applications by letting them predict and seamlessly adapt to future user needs and preferences. It improves business revenues by enabling effective product placement and targeted advertisements. Twitter data has been successfully applied to public health problems [36], crime prevention [44], and emergency response [22]. These beneficial uses have generated a big data frenzy, with Web services aggressively pursuing new ways to acquire and commercialize it.

Despite its innovative potential, the personal data frenzy has transformed the Web into an opaque and privacy-insensitive environment. Web services accumulate data, exploit it for varied and undisclosed purposes, retain it for extended periods of time, and possibly share it with others – *all without the data owner's knowledge or consent*. Who has what data, and for what purposes is it used? Are the uses in the data owners' best interests? Does the service adhere to its own privacy policy? How long is data used after its owner deletes it? Who shares data with whom?

At present, users lack answers to these questions, and investigators (such as FTC agents, journalists, or researchers) lack robust tools to track data in the ever-changing Web to provide the answers. Left unchecked, the exciting potential of big data threatens to become a breeding ground for data abuses, privacy vulnerabilities, and unfair or deceptive business practices. Examples of such practices have begun to surface. In a recent incident, Google was found to have used institutional emails from ad-free Google Apps for Education to target ads in users' personal accounts [18, 37]. MySpace was found to have violated its privacy policy by leaking personally identifiable information to advertisers [25]. Several consumer sites, such as Orbitz and Staples, were found to have adjusted their product pricing based on user location [29, 43]. And Facebook's 2010 ad targeting was shown to be vulnerable to micro-targeted ads specially crafted to reveal a user's private profile data [23].

To increase transparency and provide checks and balances on data abuse, we argue that new, robust, and versatile tools are needed to effectively track the use of personal data on the Web. Tracking data in a *controlled environment*, such as a modified operating system, language, or runtime, is an old problem with a well-known solution: taint tracking systems [12, 16, 7, 48]. However, is it possible to track data in an *uncontrolled environment*, such as the Web? Can robust, generic mechanisms assist in doing so? What kinds of data uses are trackable and what are not? How would the mechanisms scale with the amount of data being tracked?

As a first step toward answering these questions, we built *XRay*, a personal data tracking system for the Web.

XRay correlates designated data *inputs* (be they emails, searches, or visited products) with data *outputs* results (such as ads, recommended products, or prices). Its correlation mechanism is service agnostic and easy to instantiate, and it can track data use within and across services. For example, it lets a data owners track how their emails, Google+, and YouTube activities are used to target ads in Gmail.

At its core, XRay relies on a *differential correlation* mechanism that pinpoints targeting by comparing outputs in different accounts with similar, but not identical, subsets of data inputs. To do so, it associates with every personal account a number of *shadow accounts*, each of which contains different data subsets. The correlation mechanism uses a simple Bayesian model to compute and rank scores for every data input that may have triggered a specific output. Intuitively, if an ad were seen in many accounts that share a certain email, and never in accounts that lack that email, then the email is likely to be responsible for a characteristic that triggers the ad. The email's score for that ad would therefore be high. Conversely, if the ad were seen rarely in accounts with or lacking that email, that email's score for this ad would be low.

Constructing a practical auditing system around differential correlation raises significant challenges. Chief among them is scalability with the number of data items. Theoretically, XRay requires a shadow account for each combination of data inputs to accurately pinpoint correlation. That would suggest an *exponential number of accounts*! Upon closer examination, however, we find that a few realistic assumptions and novel mechanisms let XRay reach high precision and recall with only a *logarithmic number of accounts in number of data inputs*. We deem this a major new result for the science of tracking data-targeting on the Web.

We built an XRay prototype and used it to correlate Gmail ads, Amazon product recommendations, and YouTube video suggestions to user emails, wish lists, and previously watched videos, respectively. While Amazon and YouTube provide detailed explanations of their targeting, Gmail does not, so we manually validated associations. For all cases, XRay achieved 80-90% precision and recall. Moreover, we integrated our Gmail and YouTube prototypes so we could track cross-service ad targeting. Although several prior measurement studies [10, 47, 21, 20, 31] used methodologies akin to differential correlation, we believe we are the first to build a generic, service agnostic, and scalable tool based on it. Overall, we make the following contributions:

1. The first general, versatile, and open system to track arbitrary personal Web data use by uncontrolled services. The code is available from our Web page `https://xray.cs.columbia.edu/`.
2. The first in-depth exploration into the scalability challenges of tracking personal data on the Web.
3. The design and implementation of robust mechanisms to address scaling, including data matching.
4. System instantiation to track data on three services (Gmail, Amazon, YouTube) and across services (YouTube to Gmail).
5. An evaluation of our system's precision and recall on Gmail, Amazon, and YouTube. We show that XRay is accurate and scalable. Further, it reveals intriguing practices now in use by Web services and advertisers.

2 Motivation

This paper lays the algorithmic foundations for a new generation of scalable, robust, and versatile tools to lift the curtain on how personal data is being targeted. We underscore the need for such tools by describing potential usage scenarios inspired by real-life examples (§2.1). We do this not to point fingers at specific service providers; rather, we aim to show the many situations where transparency tools would be valuable for end-users and auditors alike. We conclude this section by briefly analyzing how current approaches fail to address these usage scenarios (§2.2).

2.1 Usage Scenarios

Scenario 1: Why This Ad? Ann often uses her Gmail ads to discover new retail offerings. Recently, she discussed her ad-clicking practices with her friend Tom, a computer security expert. Tom warned her about potential privacy implications of clicking on ads without knowing what data they target. For example, if she clicks on an ad targeting the keyword "gay" and then authenticates to purchase something from that vendor, she is unwittingly volunteering potentially sensitive information to the vendor. Tom tells Ann about two options to protect her privacy. She can either disable the ads altogether (using a system like AdBlock [1]), or install the XRay Gmail plugin to uncover targeting against her data. Unwilling to give up the convenience of ads, Ann chooses the latter. XRay clearly annotates the ads in the Gmail UI with their target email or combination, if any. Ann now inspects this targeting before clicking on an ad and avoids clicking if highly sensitive emails are being targeted.

Scenario 2: They're Targeting *What*? Bob, an FTC investigator, uses the XRay Gmail plugin for a different purpose: to study sensitive-data targeting practices by advertisers. He suspects a potentially unfair practice whereby companies use Google's ad network to collect sensitive information about their customers. Therefore, Bob creates a number of emails containing keywords such as "cancer," "AIDS," "bankruptcy," and "unemployment." He refreshes the Gmail page many times, each time recording the targeted ads and XRay's explanations

for them. The experiment reveals an interesting result: an online insurance company, TrustInUs.com, has targeted multiple ads against his illness-related emails. Bob hypothesizes that the company might use the data to set higher premiums for users reaching their site through a disease-targeted ad. He uses XRay results as initial evidence to open an investigation of TrustInUs.com.

Scenario 3: What's With The New Policy?[1] Carla, an investigative journalist, has set up a watcher on privacy policies for major Web services. When a change occurs, the watcher notifies her of the difference. Recently, an important sentence in Google's privacy policy has been scrapped:

> *If you are using Google Apps (free edition), email is scanned so we can display conceptually relevant advertising in some circumstances. ~~Note that there is no ad-related scanning or processing in Google Apps for Education or Business with ads disabled.~~*

To investigate scientifically whether this omission represents a shift in implemented policy, she obtains institutional accounts, connects them to personal accounts, and uses XRay to detect the correlation between emails in institutional accounts and ads in corresponding personal accounts. Finding a strong correlation, Carla writes an article to expose the policy change and its implications.

Scenario 4: Does Delete Mean Delete? Dan, a CS researcher, has seen the latest news that Snapchat, an ephemeral-image sharing Website, does not destroy users' images after the requested timeout but instead just unlinks them [41]. He wonders whether the reasons for this are purely technical as the company has declared (e.g., flash wearing levels, undelete support, spam filtering) [39, 38] or whether these photos, or metadata drawn from them, are mined to target ads or other products on the Website. The answer will influence his decision about whether to continue using the service. Dan instantiates XRay to track the correlation between his expired Snapchat photos and ads.

2.2 Alternative Approaches

The preceding scenarios illustrate the importance of transparency in protecting privacy across a range of use cases. *We need robust, generic auditing tools to track the use of personal data at fine granularity (e.g., individual emails, photos) within and across arbitrary Web services.* At present, no such tools exist, and the science of tracking the use of personal Web data at a fine grain is largely non-existent.

[1] In Feb. 2014, it was revealed based on court documents that Google could have used institutional emails to target ads in personal accounts [18]. In May 2014, Google committed to disable that feature [30]. Scenario 3 presents an XRay-based approach to investigate the original allegation.

Existing approaches can be broadly classified in two categories: *protection tools*, which prevent Web services' acquisition or use of personal data, and (2) *auditing tools*, which uncover Web services' acquisition or use of personal data. We discuss these approaches next; further related work is in §9.

Protection Tools. A variety of protection tools exist [11, 35, 1, 49]. For example, Ann could disable ads using an ad blocker [1]. Alternatively, she could encrypt her emails, particularly the sensitive ones, to prevent Google from using them to target ads. Dan could use a self-destructing data system, such as Vanish [14], to ensure the ephemerality of his Snapchat photos.

While we encourage the use of protection tools, they impose difficult tradeoffs that make them inapplicable in many cases. If Ann blocks all her ads, she cannot benefit from those she might find useful; if she encrypts all of her emails, she cannot search them; if she encrypts only her sensitive emails, she cannot protect any sensitive emails she neglected to encrypt in advance. Similarly, if Dan encrypts his Snapchat photos, sharing them becomes more difficult. While more sophisticated protection systems address certain limitations (e.g., searchable [5], homomorphic [15, 33], and attribute-based encryption [19], or privacy-preserving advertising [42, 13]), they are generally heavyweight [15], difficult to use [45], or require major service-side changes [15, 42, 13].

Auditing Tools. Given the limitations of protection tools, transparency is gaining increased attention [47, 12, 21]. If protecting data proves too cumbersome, limiting, or unsupportive of business needs, then users should at least be able to know: (1) *who is handling their data?*, and (2) *what is it being used for?*

Several tools developed in recent years partially address the first question by revealing where personal data flows from a local device [34, 12, 8]. TaintDroid [12] uses taint tracking to detect leakage of personal data from a mobile application to a service or third-party backend. ShareMeNot [34] and Mozilla's Lightbeam Firefox add-on [27] identify third parties that are observing user activities across the Web. These systems track personal data – such as location, sensor data, Web searches, or visited sites – *until it leaves the user's device*. Once the data is uploaded to Web services, it can be used or sold without a trace. In contrast, XRay's tracking just begins: we aim to tell users how services use their data *once they have it*.

Several new tools and personalization measurement studies partially address the second question: what data is being used for [10, 47, 21, 20, 31]. In general, all existing tools are highly specialized, focusing on specific input types, outputs, or services. No general, principled foundation for data use auditing exists, that can be applied effectively to many services, a primary

motivation for this our work. For example, Bobble [47] reveals search result personalization based on user location (e.g., IP) and search history. Moreover, existing tools aim to discover only *whether* certain types of user inputs – such as search history, browsing history, IP, etc. – influence the output. None pinpoints at fine grain *which* specific input – which search query, which visited site, or which viewed product – or combination of inputs explain which output. XRay, whose goals we describe next, aims to do just that.

3 Goals and Models

Our overarching goal is to develop the core abstractions and mechanisms for tracking data within and across arbitrary Web sites. After describing specific goals (§3.1), we narrow our scope with a set of simplifying assumptions regarding the data uses that XRay is designed to audit (§3.2) and the threats it addresses (§3.3).

3.1 Goals

Three specific goals have guided XRay's design:

Goal 1: *Fine-Grained and Accurate Data Tracking.* Detect which specific data inputs (e.g., emails) have likely triggered a particular output (e.g., an ad). While coarse-grained data use information (such as Gmail's typical statement, "This ad is based on emails from your mailbox.") may suffice at times, knowing the specifics can be revelatory, particularly when the input is highly sensitive and aggressively targeted.

Goal 2: *Scalability.* Make it practical to track significant amounts of data (e.g., past month's emails). We aim to support the tracking of hundreds of inputs with reasonable costs in terms of shadow accounts. These accounts are generally scarce resource since their creation is being constrained by Web services. While we assume that users and auditors can obtain *some* accounts on the Web services they audit (e.g., a couple dozen), we strive to minimize the number required for accurate and fine-grained data tracking.

Goal 3: *Extensibility, Generality, and Self-Tuning.* Make XRay generic and easy to instantiate for many services and input/output types. Instantiating XRay to track data on new sites should be simple, although it may require some service-specific implementation of input/output monitoring. However, XRay's correlation machinery – the conceptually challenging part of a scalable auditing tool – should be turn key and require no manual tuning.

3.2 Web Service Model

These goals may appear unsurmountable. An extremely heterogeneous environment, the Web has perhaps as many data uses as services. Moreover, data mining algorithms can be complex and proprietary. How can we abstract away this diversity and complexity to design robust and generic building blocks for scalable data tracking? Fortunately, we find that certain popular classes of Web data uses lend themselves to principled abstractions that facilitate scalable tracking.

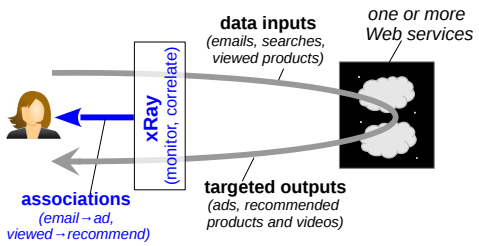


Figure 1: **XRay Conceptual View.** XRay views Web services as black boxes, monitors user *inputs* and *outputs* to/from them, and detects data use through correlation. It returns to the user or auditor *associations* of specific inputs and outputs.

Figure 1 shows XRay's simplified view of Web services. Services, and networks of services that exchange user data, are *black boxes* that receive personal data *inputs* from users – such as emails, pictures, search queries, locations, or purchases – and use them for varied purposes. Some uses materialize into *outputs* visible to users, such as ads, product or video recommendations, or prices. Others invisible to the users. XRay correlates some visible data inputs with some visible outputs by monitoring them, correlating them, and reporting strong *associations* to users. An example association is which email(s) contributed to the selection of a particular ad.

XRay relates only *strongly correlated* inputs with outputs. If an output is strongly correlated to an input (i.e., the input's presence or absence changes the output), then XRay will likely be able to detect its use. If not (i.e., the monitored input plays but a small role in the output), then it may go undetected. XRay also relates small combinations of inputs with strongly correlated outputs.

Although simple, this model efficiently addresses several types of personal data functions, including product recommendations, price discriminations, and various personalization functions (e.g., search, news). We refer to such functions generically as *targeting functions* and focus XRay's design on them.

Three popular forms of targeting are:

1. *Profile Targeting*, which leverages static or slowly evolving explicit information – such as age, gender, race, or location – that the user often supplies by filling a form. This type of targeting has been studied profusely [10, 47, 21, 20, 31]; we thus ignore it here.
2. *Contextual Targeting*, which leverages the content currently being displayed. In Gmail, this is the currently open email next to which the ad is shown. In Amazon or Youtube, the target is the product or video next to which the recommendation is shown.
3. *Behavioral Targeting*, which leverages a user's past actions. An email sent or received today can trigger an ad tomorrow; a video watched now can trig-

ger a recommendation later. Use of histories makes it harder for users to track which data is being used, a key motivation for our development of XRay.

Theoretically, our differential correlation algorithms could be applied to all three forms of targeting. From a systems perspective, XRay's design is geared towards *contextual targeting* and *a specific form of behavioral targeting*. The latter requires further attention. We observe that this broad targeting class subsumes multiple types of targeting that operate at different granularities. For example, a service could use as inputs a user's most recent few emails to decide targeting. This would be similar to an extended context. Alternatively, a service could use historical input to learn a user's coarse interests or characteristics and base its targeting on that.

XRay currently aims to disclose any targeting applied at the level of individual user data, or small combinations thereof. Our differential correlation algorithms could be applied to detect targeting that operates on a coarser granularity. However, the XRay system itself would require significant changes. Unless otherwise noted, we use *behavioral targeting* to denote the restricted form of behavioral targeting that XRay is designed to address. We formalize these restrictions in §4.2.

3.3 Threat Model

To further narrow our problem's scope, we introduce threat assumptions. We assume that data owners (users and auditors) are trusted and do not attempt to leverage XRay to harm Web services or the Web ecosystem. While they trust Web services with their data, they wish to better understand how that data is being used. Data owners are thus assumed to upload the data in clear text to the Web services.

The threat models relevant for Web services depend on the use case. For example, Scenarios 1 and 2 in §2.1 assume Google is trusted, but its users wish to understand more about how advertisers target them through its ad platform. In contrast, in Scenarios 3 and 4, investigators may have reason to believe that Web services might intentionally frustrate auditing.

This paper assumes an *honest-but-curious* model for Web services: they try to use private data for financial or functional gains, but they do not try to frustrate our auditing mechanism, e.g., by identifying and disabling shadow accounts. The service might attempt to defend itself against more general types of attacks, such as spammers or DDoS attacks. For example, many Web services constrain the creation of accounts so as to limit spamming and false clicks. Similarly, Web services may rate limit or block the IPs of aggressive data collectors. XRay must be robust to such inherent defenses. We discuss challenges and potential approaches for stronger adversarial models in §7.

4 The XRay Architecture

XRay's design addresses the preceding goals and assumptions. For concreteness, we draw examples from our three XRay instantiations: tracking email-to-ad targeting association within Gmail, attributing recommended videos to those already seen on YouTube, and identifying products in a wish list that generate a recommendation on Amazon.

4.1 Architectural Overview

XRay's high-level architecture (Figure 2) consists of three components: (1) a *Browser Plugin*, which intercepts tracked inputs and outputs to/from an audited Web service and gives users visual feedback about any input/output associations, (2) a *Shadow Account Manager*, which populates shadow accounts with inputs from the plugin and collects outputs (e.g., ads) for each shadow account, and (3) the *Correlation Engine*, XRay's core, which infers associations and provides them to the plugin for visualization. While the Browser Plugin and Shadow Account Manager are *service specific*, the Correlation Engine, which encapsulates the science of Web-data tracking, is *service agnostic*. After we describe each component, we focus on the design of the Correlation Engine.

Browser Plugin. The Browser Plugin intercepts designated inputs and outputs (i.e., *tracked inputs/outputs*) by recognizing specific DOM elements in an audited service's Web pages. Other inputs and outputs may not be tracked by XRay (i.e., *untracked inputs/outputs*). The decision of what to track belongs to an investigator or developer who instantiates XRay to work on a specific service. For example, we configure the XRay Gmail Plugin to monitor a user's emails as inputs and ads as outputs. When the Plugin gets a new tracked input (e.g., a new email), it forwards it both to the service and to the Shadow Account Manager. When the Plugin gets a new tracked output (e.g., an ad), it queries the Correlation Engine for associations with the user's tracked inputs (message `get_assoc`).

Shadow Account Manager. This component: (1) populates the shadow accounts with subsets of a user account's tracked inputs (denoted D_i), and (2) periodically retrieves outputs (denoted O_k) from the audited service for each shadow account. Both functions are service specific. For Gmail, they send emails with SMTP and call the ad API. For YouTube, they stream a video and scrape recommendations, and for Amazon, they place products in wish lists and scrape recommendations. The complexity of these tasks depends on the availability of APIs or the stability of a service's page formats. Outputs collected from the Web service are placed into a *Correlation Database* (DB), which maps shadow accounts to their input sets and output observations. Figure 2 shows a par-

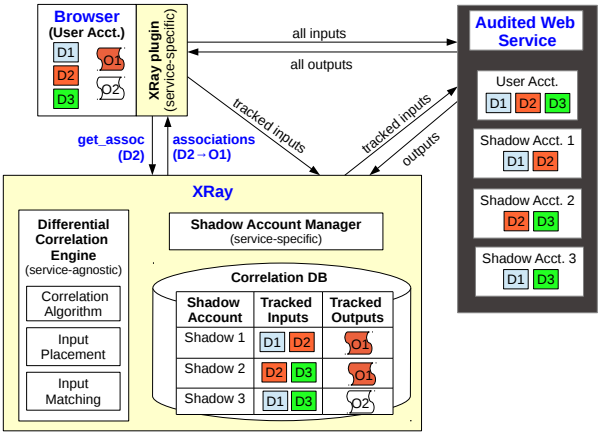


Figure 2: **The XRay Architecture.**

ticular assignment of tracked inputs across three shadow accounts. For example, Shadow 1 has inputs D_1 and D_2. The figure also shows the outputs collected for each shadow account. Output O_1 appears in Shadows 1 and 2 but not in 3; output O_2 appears in Shadow 3 only.

Differential Correlation Engine. This engine, XRay's service-agnostic "brain," leverages the data collected in the Correlation DB to infer input/output associations. When new outputs from shadow accounts are added into the Correlation DB, the engine attempts to diagnose them using a *Correlation Algorithm*. We developed several such algorithms and describe them in §4.3. This process, potentially time-consuming process, is done as a background job, asynchronously from any user request. In Figure 2, differential correlation might conclude that D_2 triggers O_1 because O_1 appears consistently in accounts with that D_2. It might also conclude that O_2 is *untargeted* given inconsistent observations. The engine saves these associations in the Correlation DB.

When the plugin makes a `get_assoc` request, the Correlation Engine looks up the specified output in its DB and returns any pre-computed association. If no output is found, then the engine replies *unknown* (e.g., if an ad never appeared in any shadow account or there is insufficient information). Periodic data collection, coupled with an online update of correlation model parameters, minimizes the number of unknown associations. Our experience shows that collecting shadow account outputs in Gmail every ten hours or so yielded few unknown ads.

While the preceding example is simple, XRay can handle complex challenges occurring in practice. First, outputs are never consistently seen across all shadow accounts containing the input they target. We call this the *limited-coverage* problem; XRay handles it by placing each data input in more shadow accounts. Second, an output may have been triggered by one of several targeted inputs (e.g., multiple emails on the same topic may cause related ads to appear), a problem we refer to as *overlapping-inputs*. This exacerbates the number of accounts needed, since it diminishes the differential signal we receive from them. XRay uses robust, service-agnostic mechanisms and algorithms to match overlapping inputs, place them in the same accounts, and detects their use as a group.

Organization. The remainder of this section describes the Differential Correlation Engine. After constructing it for Gmail, we applied it as-is for Amazon and YouTube, where it achieved equally high accuracy and scalability despite observable differences in how targeting works on these three services. After establishing notations and formalizing our assumptions (§4.2), we describe multiple correlation algorithms, which build up to our self-tuning correlation algorithm that made this adaptation convenient (§4.3). §4.4 describes our input matching.

4.2 Notation and Assumptions

We use f to denote the black-box function that represents the service (e.g., Gmail) associating inputs D_is (e.g., the emails received and sent) to targeted outputs O_ks (e.g., ads). Other inputs are either ignored by XRay, known only to the targeting system, or under no known control. We assume they are independent or fixed, captured in the randomness of f.

We assume that f decides targeting using: (1) a single input (e.g., show O_k if D_4 is in the account), (2) a conjunctive combination of inputs (e.g., show O_k if D_5 *and* D_8 are in the account), or (3) a disjunctive combination of the previous (e.g., show O_k if (D_5 *and* D_8) are in the account *or* if D_4 is in the account). We refer to conjunctive and disjunctive combinations as AND and OR combinations, respectively, and assume that their is bounded by a maximum *input size*, r. This corresponds to the preceding definition of behavioral targeting from §3.2. Contextual targeting will always be a single-input (size-one) combination.

Our goal is to decide whether f produced each output O_k as a reaction to a bounded-size combination of the D_is. We define as *untargeted* any ad that is not targeted against any combination of D_is, though in reality the ad could be targeted against untracked inputs. We denote untargeting as $D_\emptyset$, meaning that the ad is targeted against the "void" email. Our algorithms compute the most likely combination from the N inputs that explains a particular set of observations, $\vec{x}$, obtained by XRay.

We define three probabilities upon which our algorithms and analyses depend. First, the *coverage*, p_{in}, is the probability that an account j containing the input D_i targeted by a particular ad, will see that ad at least once. Second, an account j' lacking input D_i will see the ad with a smaller probability, p_{out}. Third, if the ad is not behaviorally targeted, it will appear in each account with the same probability, $p_\emptyset$. We assume that $p_{\text{in}}, p_\emptyset, p_{\text{out}}$ are

constant across all emails, ads, and time, and that p_{out} is strictly smaller than p_{in} (bounded noise hypothesis).

Finally, we consider all outputs to be independent of each other across time. §8 discusses the implications.

4.3 Correlation Algorithms

A core contribution of this paper is our service-agnostic, self-tuning differential correlation algorithm, which requires only a logarithmic number of shadow accounts to achieve high accuracy. We wished not only to validate this result experimentally, but also to prove it theoretically in the context of our assumptions. This section constructs the algorithm in steps, starting with a naïve polynomial algorithm that illustrates the scaling challenges. We then define a base algorithm using set intersections and prove that it has the desired logarithmic scaling properties; it has parameters which, if not carefully chosen, can lead to poor results. We therefore extend this base algorithm into a self-tuning Bayesian model that automatically adjusts its parameters to maximize correctness.

4.3.1 Naïve Non-Logarithmic Algorithm

An intuitive approach to differential correlation is to create accounts for every combination of inputs, gathering maximum information about their behaviors. With a sufficient number of observations, one could expect to detect which accounts, and hence which subsets of inputs, target a particular ad. Unfortunately, this method requires a number of accounts that grows *exponentially* as the number of items N to track grows. When restricting the size of combinations to r, as we do in XRay, the number of accounts needed is *polynomial* (in $O(N^r)$), or *linear* if we study unique inputs only. Even a linear number of accounts in the number N of inputs remains impractical to scale to large input sizes (e.g., a mailbox).

4.3.2 Threshold Set Intersection

We now show that it is possible to infer behavioral targeting using no more than a *logarithmic* number of accounts as a function of the number of inputs. Specifically, we prove the following theorem:

Theorem 1 *Under §4.2 assumptions, for any $\varepsilon > 0$ there exists an algorithm that requires $C \times \ln(N)$ accounts to correctly identify the inputs of a targeted ad with probability $(1-\varepsilon)$. The constant C depends on ε and the maximum size of combinations r ($O(r2^r \log(\frac{1}{\varepsilon}))$).*

To demonstrate the theorem, we define the *Set Intersection Algorithm* and prove that it has the correctness and scaling properties specified in the theorem. Given that outputs will appear more often in accounts containing the targeting inputs, the core of the algorithm is to determine the set of inputs appearing in the highest number of accounts that also see a given ad. This paper describes a basic version of the algorithm that makes some simplifying assumptions and provides a brief proof sketch. The detailed proof and complete algorithm are described in our technical report [26].

```
// Set Intersection Algo:
// Runs with each collected ad.
In: Output O_k (e.g. an ad).
Params: MIN_ACTIVE_ACCTS, THRESHOLD.
Out: Targeted input combination.
// Step 1: Compute active accounts.
A_k = the accounts that see ad O_k.
if |A_k| < MIN_ACTIVE_ACCTS
    return ∅
end
// Step 2: Create input combination hypothesis.
targeted_set = ∅
foreach input D_i do
    if (number of A_k containing D_i) / |A_k| > THRESHOLD
        targeted_set += D_i
    end
end
// Step 3: Verify it is a real combination.
if (number of A_k containing entire targeted_set) / |A_k| < THRESHOLD
    return ∅
end
// targeted_set triggered the output.
return targeted_set
```

Figure 3: **The Set Intersection Algorithm.** Can be proven to predict targeting correctly under certain assumptions with a logarithmic number of accounts.

Algorithm. The algorithm relies on a randomized placement of inputs into shadow accounts, with some redundancy to cope with imperfect coverage. We thus pick a probability, $0 < \alpha < 1$, create $C\ln(N)$ shadow accounts, and place each input D_i randomly into each account with probability α. Figure 3 shows the Set Intersection algorithm for a set of observations, $\vec{x}$. Given an output O_k collected from the user account, we compute the set of *active accounts*, A_k, as those shadow accounts that have seen the output (Step 1). We then compute the set of inputs that appear in at least a threshold fraction of active accounts; this set is our candidate for the combination being targeted by the ad (Step 2). Finally, we check that the entire combination is in a threshold fraction of the active accounts (Step 3). Theoretically, we prove that there exists a threshold for which the algorithm is arbitrarily correct with the available $C\ln(N)$ accounts. Practically, this threshold must be tuned experimentally to achieve good accuracy on every service – a key reason for our Bayesian enhancement in §4.3.3.

Correctness Proof Sketch. The proof shows that if there were targeting, every non-targeting input would have a vanishingly small probability to be in a significant fraction of the active accounts. Let us call S the set of inputs

```
// Bayesian Prediction Alg:
// Runs with each collected ad.
In: Output O_k (e.g. an ad).
Out: Targeted input.
// Compute probabilities.
foreach input D_i do
  P[D_i|x] = bayes(P[x|D_i])
end
// Compute untargeted prob.
P[D_∅|x] = bayes(P[x|D_∅])
// Return event with max prob.
return D_i with max P[D_i|x]
```

```
// Parameter Learning Alg:
// Runs periodically.
// Initialize params (arbitrary).
p_in = .7, p_out = .01, p_∅ = .1
do
  foreach output O_k do
    Run Bayesian Prediction.
  end
  Update p_in, p_out, p_∅
    from predictions.
until p_in, p_out, p_∅ converge
end
```

Figure 4: **Bayesian Correlation.** Left: Bayesian prediction algorithm for behavioral targeting. Right: typical iterative inference process to learn parameters.

contained in a significant fraction of the active accounts. Without targeting, these inputs would be present in the accounts by mere chance. Since inputs are independently distributed into the accounts, we show that the probability of S not being empty decreases exponentially with the number of active accounts (through Chernoff bounds). With targeting, we show that with high probability no other input than the explaining combination is in S, because of the bounded noise hypothesis. Appendix A.2 provides further proof details.

The proofs and algorithm included in this paper work only for conjunctive combinations (e.g., D_1 and D_2, see §4.2). The theory, however, can be extended to disjunctive combinations (e.g., (D_1 and D_2) or D_5), but the algorithm for detecting such combinations is more complex and relies on a recursive argument: if we find one combination from the disjunction, then the active accounts that include this combination define a context where the combination appears non-targeting because it is everywhere. If we recursively apply our algorithm in this context, we can detect the second combination in the disjunction, then the third, etc (see technical report [26]).

4.3.3 Self-Tuning Bayesian Algorithm

The Set Intersection algorithm provides a good theoretical foundation; however, it requires parameters be tuned and applies only to behavioral targeting, not contextual targeting. Thus, we include in XRay a more robust, self-tuning version that leverages a Bayesian algorithm to adjust parameters automatically through iterated inference. Our algorithm relies on three models: one that predicts behavioral targeting, one that predicts contextual targeting, and one that combines the two.

Behavioral Targeting. The Bayesian behavioral targeting model uses the same random assignment as the Set Intersection algorithm, and it leverages the same information from the shadow account observations, $\vec{x}$. It counts the observations x_j of ad O_k in an account j as a binary signal: if the ad has appeared at least once in account j, we count it once; otherwise we do not count it. Briefly, the Bayesian model is a simple generative model that simulates the audited service given some targeting associations (e.g., D_i triggers O_k). It computes the probability for this model to generate the outputs we do observe for every targeting association. The most likely association will be the one XRay returns.

In more detail if the ad were targeted towards D_i, then an account j containing D_i would see this ad at least once with a *coverage* probability p_{in}; otherwise, it would miss it with probability $(1-p_{\text{in}})$. An account j' without input D_i would see the ad with a smaller probability, p_{out}, missing it with probability $(1-p_{\text{out}})$. If the ad were not behaviorally targeted, it would appear in each account with the same probability, $p_\emptyset$. If we define A_k as the set of active accounts that have seen the ad, and A_i as the set of accounts that contain email D_i, then we have the following definitions for the probabilities:

$$\begin{aligned}\mathbb{P}[\vec{x}|\, D_i] = &\quad (p_{\text{in}})^{|A_i\cap A_k|}(1-p_{\text{in}})^{|A_i\cap \bar{A}_k|}\\ &\times (p_{\text{out}})^{|\bar{A}_i\cap A_k|}(1-p_{\text{out}})^{|\bar{A}_i\cap \bar{A}_k|},\\ \mathbb{P}[\vec{x}|\, D_\emptyset] = &\quad (p_\emptyset)^{|A_k|}(1-p_\emptyset)^{|\bar{A}_k|},\end{aligned}$$

where $D_\emptyset$ designates the untargeted prediction.

The preceding formula has an interesting interpretation that is visible if placed in the equivalent form:

$$\begin{aligned}\mathbb{P}[\vec{x}|\, D_i] = &\quad (p_{\text{in}})^{|A_k|}(1-p_{\text{out}})^{|\bar{A}_k|}\\ &\times \left(\frac{1-p_{\text{in}}}{1-p_{\text{out}}}\right)^{|A_i\cap\bar{A}_k|}\left(\frac{p_{\text{out}}}{p_{\text{in}}}\right)^{|\bar{A}_i\cap A_k|}\end{aligned}$$

From the point of view of the event D_i, an account found in $A_i\cap\bar{A}_k$ is a false positive (an ad was expected but was not shown). This should lower the probability, especially when the *coverage* p_{in} is close to 1. Inversely, an account found in $\bar{A}_i\cap A_k$ acts as a false negative (we observed an ad where we did not expect it), which should decrease the probability, especially when p_{out} is close to 0.

These formulas let us infer the likelihood of event D_i according to Bayes' rule: $\mathbb{P}[A|\, B] = \frac{\mathbb{P}[B|A]\times\mathbb{P}[A]}{\mathbb{P}[B]}$. Figure 4 shows two algorithms. First, the prediction algorithm (left) predicts the targeting of O_k by computing the probabilities defined above, applying Bayes' rule, and returning the input with the maximum probability. Second, the parameter learning algorithm (right) computes the variables that those probabilities depend upon (p_{in}, p_{out}, and $p_\emptyset$) using an iterative process. It repeatedly runs the prediction algorithm for all outputs and re-computes p_{in}, p_{out}, and $p_\emptyset$ based on the predictions. It stops when the variables converge (i.e., their variation from one iteration to another is small).

Contextual Targeting. Contextual targeting is more straightforward since it uses content shown next to the ad. XRay also uses Bayesian inference and defines the observations as how many times ad O_k is seen next to

email D_i. Our causal model assumes imperfect coverage: if this ad were contextually targeted towards D_i, it would occur next to that email with probability $p_{\text{in}} < 1$ and next to any other email with probability p_{out}. Alternatively, if the ad were untargeted, our model predicts it would be shown next to any email with probability $p_{\emptyset}$. Hence, $\mathbb{P}[\vec{x}|D_i] = (p_{\text{in}})^{x_i}(p_{\text{out}})^{\sum_{i' \neq i} x'_i}, \mathbb{P}[\vec{x}|D_{\emptyset}] = (p_{\emptyset})^{\sum_i x_i}$. For this model, parameters are also automatically computed by iterated inference.

Composite Model (XRay). The contextual and behavioral mechanisms were designed to detect different types of targeting. To detect both types, XRay must combine the two scores. We experimented with multiple combination functions, including a decision tree and the arithmetic average, and concluded that the arithmetic average yields sufficiently good results. XRay thus defines the *composite model* that averages scores from individual models, and we demonstrate in §6.3 that doing so yields higher recall for no loss in precision.

4.4 Input Matching and Placement

Our design of differential correlation, along with our logarithmic results for random input placement, relies on the fundamental assumption that the probability of getting an ad O_1 targeted at an input D_1 in a shadow account that lacks D_1 is vanishingly small. However, when inputs attract the same ads (a.k.a., overlapping inputs), a naive input placement can contradict this assumption. Imagine a Gmail account with multiple emails related to a Caribbean trip. If placement includes Caribbean emails in every available shadow account, related ads will appear in groups of accounts with no email object in common. XRay will thus classify them as untargeted. Our Amazon experiments showed XRay's recall dropping from 97% to 30% with overlapping inputs (§6.5).

To address this problem, XRay's Input Matching module identifies similar inputs and directs the Placement Module to co-locate them in the same shadow accounts. The key challenge is to identify similar inputs. One method is to use content analysis (e.g., keywords matching), but this has limitations. First, it is not service agnostic; one needs to reverse engineer complex and ever-changing matching schemes. Second, it is hard to apply to non-textual media, such as YouTube videos.

In XRay, we opt for a more robust, systems technique rooted in the key insight that we can deduce similar inputs from contextual targeting. Intuitively, inputs that trigger similar targeting from the Web service should attract similar outputs in their context. The Input Matching module builds and compare inputs' *contextual signatures*. Contextual signature similarity is the distance between inputs (e.g., email) in a Euclidean space, where each output (e.g., ad) is a dimension. The coordinate of an email in this dimension is the number of times the ad was seen in the context of the email. XRay then forwards close inputs to the same shadow accounts. Once the placement is done, behavioral targeting against that email's group can be inferred effectively.

This input matching mechanism differs fundamentally from any content analysis technique, such as keyword matching, because it groups inputs *the same way the Web service does*.[2] It is robust and very general: we used it on both Gmail and Amazon without changing a single line of code to change.

5 XRay-based Tools

To evaluate XRay's extensibility, we instantiated it on Gmail, YouTube, and Amazon. The engine, about 3,000 lines of Ruby, was first developed for Gmail. We then extended it to YouTube and Amazon, without any changes to its correlation algorithms. We did need to do minor code re-structuring, but the experience felt turn key when integrating a new service into the correlation machinery.

Building the full toolset required non-trivial coding effort, however. Instantiating XRay for a specific Web service is a three-step process. First, the developer instantiates appropriate data models (less than 20 code lines for our prototypes). Second, she implements a service-specific shadow account manager and plugin; care must be taken not be too aggressive to avoid adversarial service reactions. While these implementations are conceptually simple, they require some coding; our Amazon and YouTube account managers were built by two graduate students new to the project, and have around 500 lines of code. Third, the developer creates a few shadow accounts for the audited service and runs a small exploratory experiment to determine the service's coverage. XRay uses the coverage to estimate the number of shadow accounts needed for a given input size. All other parameters are self-tuned at runtime.

6 Evaluation

We evaluated XRay with experiments on Gmail, Amazon, and YouTube. While Amazon and YouTube provide ground truth for their targeting, Gmail does not. We therefore manually labeled ads on Gmail and measured XRay's accuracy, as described in §6.1 and validated in §6.2. We sought answers to four questions:

Q1 How accurate are XRay's inference models? (§6.3)
Q2 How does XRay scale with input size? (§6.4)
Q3 Can input matching manage overlap? (§6.5)
Q4 How useful is XRay in practice? (§6.6)

[2] We call this method "monkey see, monkey do" because we watch how the service groups inputs and group them similarly.

Ad Keyword	Targeted Email	Detected by XRay?	XRay Scores	# Accounts & Displays
Chaldean Poetry	Like Chaldean Poetry?	Yes	0.99, 1.0	13/13, 1588/1622
Steampunk	Fan of Steampunk?	Yes	0.99, 1.0	13/13, 888/912
Cosplay	Discover Cosplay.	Yes	0.99, 1.0	13/13, 440/442
Falconry	Learn about Falconry.	Yes	0.99, 1.0	13/13, 1569/1608

Figure 5: **Self-Targeted Ads.** Fourth column shows XRay's correlation scores X, Y, (Bayesian) Behavioral and Contextual scores, respectively. Fifth column shows raw behavioral and contextual data for interpretation: X/Y, Z/T means that the ad was seen in X active accounts that contain the targeted email out of a total of Y active accounts; the ad was shown Z times in the context of the targeted email out of a total of T times.

6.1 Methodology

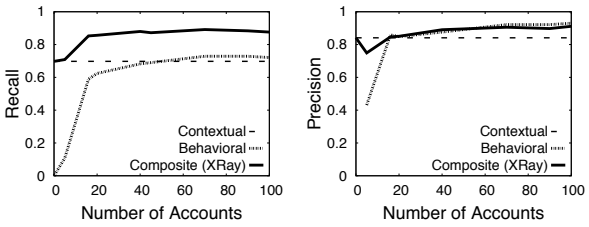
We evaluated XRay with experiments on Gmail, Amazon, and YouTube. For inputs, we created a workload for each service by selecting topics from well-defined categories relevant for that service. For Gmail and YouTube, we crafted emails and selected videos based on AdSense categories [17]; for Amazon, we selected products from its own product categories [2]. We used these categories for most of our experiments (§6.3–§6.5). We used these categories to create two types of workloads: (1) a non-overlapping workload, in which each data item belonged to a distinct category, and (2) an overlapping workload, with multiple data items per category (described in §6.5).

To assess XRay's accuracy, we needed the ground truth for associations. Amazon and YouTube provide it for their recommendations. For instance, Amazon provides a link "Why recommended?" which explicitly explains the recommendation. For Gmail, we manually labeled ads based on our personal assessment. The ads for different experiments were labeled by different people, generally project members. A non-computer scientist labeled the largest experiment (51 emails).

We evaluate two metrics: (1) *recall*, the fraction of positive associations labeled as such, and (2) *precision*, the fraction of correct associations. We define *high accuracy* as having both high recall and high precision.

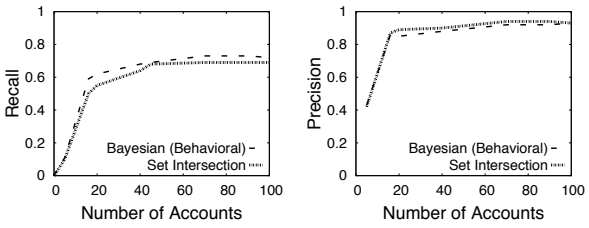
6.2 Sanity-Check Experiment

To build intuition into XRay's functioning, we ran a simple sanity-check experiment on Gmail. Recall that, unlike Amazon and YouTube, Gmail does not provide any ground truth, requiring us to manually label associations, a process that can be itself faulty. Before measuring XRay's accuracy against labeled associations, we checked that XRay can detect associations for our own ads, whose targeting we control. For this, we strayed away from the aforementioned methodology to create a highly controlled experiment. We posted four Google AdWords campaigns targeted on very specific keywords (Chaldean Poetry, Steampunk, Cosplay, and



(a) **Recall** (b) **Precision**

Figure 6: **Bayesian Model Accuracy.** Recall and precision for each of the three Bayesian models vs. shadow account number, using the Bayesian algorithm. XRay needed 16 accounts to reach the "knee" with high recall and precision.



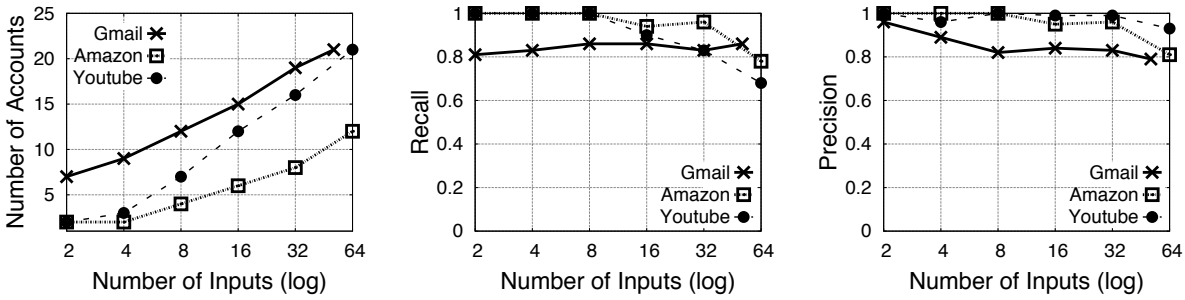
(a) **Recall** (b) **Precision**

Figure 7: **Bayesian vs. Set Intersection Comparison.** Recall and precision for detecting *behavioral* targeting with each algo.

Falconry), crafted an inbox that included one email per keyword, and used XRay to recover the associations between our ads and those emails. In total, we saw our ads 1622, 912, 442, and 1608 times, respectively, across all accounts (shadows and master). Figure 5 shows our results. After one round of ad collection (which involved 50 refreshes per email), XRay correctly associated all four ads with the targeted email. It did so with very high confidence: composite model scores were 0.99 in all cases, with very high scores for both contextual and behavioral models. The figure also shows some of the raw contextual/behavioral data, which provides intuition into XRay's perfect precision and recall in this controlled experiment. We next turn to evaluating XRay in less controlled environments, for which we use the workloads and labeling methodology described in §6.1.

6.3 Accuracy of XRay's Inference Models (Q1)

To assess the accuracy of XRay's key correlation mechanisms (Bayesian behavioral, contextual, and composite), we measured their recall and precision under non-overlapping workloads. Figures 6(a) and 6(b) show how these two metrics varied with the number of shadow accounts for a 20-email experiment on Gmail. The results indicate two effects. First, both contextual and behavioral models were required for high recall. Of the 193 distinct ads seen in the user account, 121 (62%) were targeted, and XRay found 109 (90%) of them, a recall we deem high. Of the associations XRay found, 37% were found by only one of the models: 15



(a) **Scalability with Input Size** (b) **Recall with Input Size** (c) **Precision with Input Size**

Figure 8: **Scalability.** (a) Number of accounts required to achieve the knee accuracy for varied numbers of inputs. (b), (c) Recall/precision achievable with the number of accounts in (a). Behavioral uses the Bayesian algorithm.

by the contextual model only, and 24 by the behavioral model only. Thus, both models were necessary, and composing them yielded high recall. Our Amazon and YouTube experiments (which provide ground truth) yielded very similar results: on a 20-input experiment, we reached over 90% recall and precision with only 8 and 12 accounts, respectively.

Second, the composite model's recall exhibited a knee-shaped curve for increasing shadow account numbers, with a rapid improvement at the beginning and slow growth thereafter. With 16 accounts, XRay exceeded 85% recall; increasing the number of accounts to 100 yielded a 1.9% improvement. Precision also remained high (over 84%) past 16 accounts. We define the *knee* as the minimum number of accounts needed to reap most of the achievable recall and precision.

We also wished to compare the accuracy of the Bayesian algorithm, which conveniently self-tunes its parameters, to the parameterized Set Intersection algorithm. We manually tuned the latter as best as we could. Figures 7(a) and 7(b) show the recall and precision for detecting behavioral targeting with the two methods for a non-overlapping workload. The two algorithms performed similarly, with the Bayesian staying within 5% of the manually tuned algorithm. We also tested the algorithms on an Amazon dataset, and using a version of the Set Intersection algorithm with empirical optimizations. The conclusion holds: the Bayesian algorithm, with self-tuned parameters, performs as well as the Set Intersection technique with manually tuned parameters. We focus the remainder of this evaluation on the Bayesian algorithm.

6.4 Scalability of XRay with Input Size (Q2)

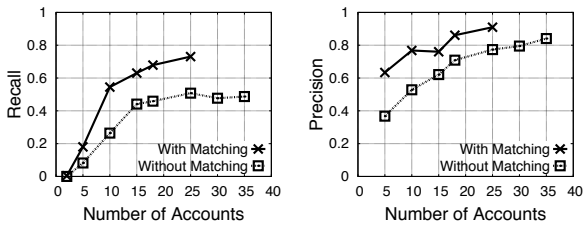
A main contribution of this paper is the realization that, under certain assumptions, the number of accounts needed to achieve high accuracy for XRay scales logarithmically with the number of tracked inputs. We have proven that under certain assumptions, the Set Intersection algorithm scales logarithmically. This theoretical result is hard to extend to the Bayesian algorithm, so we evaluated it experimentally by studying three metrics with growing input size: the number of accounts required to reach the recall knee and the value of recall/precision at this knee. Figures 8(a), 8(b) and 8(c) show the corresponding results for Gmail, YouTube and Amazon. For Gmail, the number of accounts necessary to reach the knee increased less than 3-fold (from 8 to 21) as input size increased more than 25-fold (from 2 to 51). For Amazon and YouTube, the increases in accounts were 6- and 8-fold respectively, for a 32-fold increase in input size. In general, the roughly linear shapes of the log-x-scale graphs in Figure 8(a) confirm the logarithmic increase in the number of accounts required to handle different inputs. Figure 8(b) and 8(c) confirm that the "knee number" of accounts achieved high recall and precision (over 80%).

What accounts for the large gap between the number of accounts needed for high accuracy in Gmail versus Amazon? For example, tracking a mere two emails in Gmail required 8 accounts, while tracking two viewed products in Amazon needed 2 accounts. The distinction corresponds to the difference in coverage exhibited by the two services. In Gmail, a targeted ad was typically seen in a smaller fraction of the relevant accounts compared to a recommended product in Amazon. XRay adapted its parameters to lower coverage automatically, but it needed more accounts to do so.

Overall, these results confirm that our theoretical scalability results hold for real-world systems given carefully crafted, non-overlapping input workloads. We next investigate how more realistic overlapping input workloads challenge the accuracy of our theoretical models and how input matching – a purely systems technique – helps address this challenge.

6.5 Input Matching Effectiveness (Q3)

To evaluate XRay's accuracy with overlapping inputs, we infused our workloads with multiple items from the same category. (e.g., multiple emails targeting the



(a) **Recall** (b) **Precision**

Figure 9: **Input Matching effectiveness.** Behavioral (Bayesian) recall and precision in Gmail with overlapping inputs, with and without Matching.

same topic on Gmail and multiple products in the same category in Amazon). For the Gmail experiments, we (as users) could not tell when Gmail targeted a specific email from a group of similar emails. We therefore ran two different types of experiments. First, a controlled, albeit unrealistic, one for Gmail. We replicated various emails *identically* in a user's inbox: 1 email was replicated 4 times, 2 emails 3 times, 4 emails 2 times, and 12 were single, for a total of 30 emails. This end-of-a-spectrum workload demonstrates how matching works ideally. XRay matched *all* redundant emails correctly. More importantly, Figures 9(a) and 9(b) show XRay's precision/recall with and without matching-aware placement for XRay's behavioral model, the only model improved by matching. Without input matching, XRay struggled to find differential signals: even with 35 shadow accounts for a 30-email experiment, recall was only 48%. With input matching, XRay's correlation model drew a stronger signal from each account and attained close to 70% recall for 16 accounts.

Second, for Amazon, we created a more realistic overlapping workload by selecting three *distinct* products in each of six product categories (e.g., from the Outdoor & Cycling category, we selected a helmet, pedals, and shoes). With a total workload of 18 products, XRay's input matching matched all but one item (shoes) into its correct group. With the new grouping, XRay's recall improved by a factor of 3 (from 30% to 93%) compared to the no-matching case for 18 products with 10 accounts; precision was 2.6 times higher (from 34% to 88%).

These results demonstrate that XRay's matching scheme is both portable across Web services and essential for high accuracy with overlapping workloads.

6.6 Anecdotal Use Experience (Q4)

To gain intuition into XRay's practical value, we ran a small-scale, anecdotal experiment that fished for Gmail ads targeted against a few specific topics. We created emails focused on topics such as cancer, Alzheimer, depression, HIV, race, homosexuality, pregnancy, divorce, and debt. Each email consisted of keywords closely related to one topic (e.g., the depression-related email included *depression*, *depressed*, and *sad*; the homosexuality email included *gay*, *homosexual*, and *lesbian*). We then launched XRay's Gmail ad collection and examined the targeting associations. We acknowledge that a much larger-scale experiment is needed to reach statistically-meaningful conclusions. Hence, we relate our experience by example.

Topic	Targeted Ads	XRay Scores	# Accounts & Displays
Alzheimer	Black Mold Allergy Symptoms? Expert to remove Black Mold.	0.99, 0.05	9/9, 61/198
	Adult Assisted Living. Affordable Assisted Living.	0.99, 0.99	8/8, 12/14
Cancer	Ford Warriors in Pink. Join The Fight.	0.96, 0.98	9/9, 1022/1106
	Rosen Method Bodywork for physical or emotional pain.	0.98, 0.05	7/7, 24/598
Depression	Shamanic healing over the phone.	0.99, 0.99	16/16, 117/117
	Text Coach - Get the girl you want and Desire.	0.93, 0.04	7/7, 31/276
African American	Racial Harassment? Learn your rights now.	0.99, 0.2	10/10, 851/5808
	Racial Harassment, Hearing racial slurs?	0.99, 0.2	10/10, 627/7172
Homosexuality	SF Gay Pride Hotel. Luxury Waterfront.	0.99, 0.1	9/9, 50/99
	Cedars Hotel Loughborough, 36 Bedrooms, Restaurant, Bar.	0.96, 1.0	8/8, 36/43
Pregnancy	Find Baby Shower Invitations. Get Up To (60% Off) Here!	0.99, 1.0	9/9, 22/22
	Ralph Lauren Apparel. Official Online Store.	0.99, 0.6	10/10, 85/181
	Clothing Label-USA. Best Custom Woven Labels.	0.99, 1.0	9/9, 14/14
	Bonobos Official Site, Your Closet Will Thank You.	0.99 0.99	9/9 64/71
Divorce	Law Attorneys specializing in special needs kids education.	0.99, 0.99	9/9, 635/666
	Cerbone Law Firm, Helping Good People Thru Bad Times	0.99, 1.0	10/10, 94/94
Debt	Take a New Toyota Test Drive, Get a $50 Gift Card On The Spot.	0.99, 0.9	7/7, 58/65
	Great Credit Cards Search. Apply for VISA, MasterCard...	0.99, 0.0	9/9, 151/2358
	Stop Creditor Harassment, End the Harassing Calls.	0.99, 0.96	8/8, 256/373

Figure 10: **Example of Targeted Ads.** Columns three and four show the same data as columns four and five in Figure 5.

Figure 10 shows ads that XRay associated with each topic, with its confidence scores. Conservatively, we only consider ads with high scores. We make two observations. First, our small-scale experiment confirms that it is possible to target sensitive topics in users' inboxes. All disease-related emails, except for the HIV one, are strongly correlated with a number of ads. A "Shamanic healing" ad appears exclusively in accounts containing the depression-related email, and many times in its context; ads for assisted living services target the Alzheimer email; and a Ford campaign to fight breast cancer targets the cancer email. Race, homosexuality, pregnancy, divorce, and debt also attract plenty of ads. For example, the pregnancy email is strongly targeted by an ad for baby-shower invitations (shown in the figure), maternity- and lactation-related ads (not shown), and, interestingly, a number of ads for general-purpose clothing

(shown). As another example, the debt email is strongly targeted by a car dealership ad that entices the targeted users to take a Toyota test drive using a $50 gift offering. Discussing the morality of targeting such sensitive topics is beyond our statute, however we believe that the lack of transparency, coupled with sensitive-topic targeting, opens users to subtle dangers, a topic we discuss next.

Second, for many ads, the association with the targeted email is not obvious at all. Nothing in the "Shamanic healing" ad suggests targeting against depression; nothing in the general-purpose clothing ads suggest targeting against pregnancy; and nothing in the "Cedars hotel" ad suggests an orientation toward the homosexuality email. If no keyword in the ad suggests relation with sensitive topics, a user clicking on the ad may not realize that they could be disclosing private information to advertisers. Imagine an insurance company wanted to gain insight into pre-existing conditions of its customers before signing them up. It could create two ad campaigns – one that targets cancer and another youth – and assign different URLs to each campaign. It could then offer higher premium quotes to visitors who come through the cancer-related ads to discourage them from signing up while offering lower premium quotes to those who come through youth-related ads. We believe that the potential for this attack illustrates the urgent need for increased transparency in ad targeting.

6.7 Summary

Our evaluation results show that XRay supports fine-grained, accurate data tracking in popular Web services, scales well with the size of data being tracked, is general and flexible enough to work efficiently for three Web services, and robustly uses systems techniques to discover associations when ad contents provide no indication of them. We next discuss how XRay meets its last goal: robustness against honest-but-curious attackers.

7 Security Analysis

As stated in §3.3, two threat models are relevant for XRay and applicable to different use cases. First, an *honest-but-curious* Web service does not attempt to frustrate XRay, but it could incorporate defenses against typical Web attacks, such as DDoS or spam, that might interfere with XRay's functioning. Second, a *malicious service* takes an adversarial stand toward XRay, seeking to prevent or otherwise disrupt its correlations. Our current XRay prototype is robust against the former threat and can be extended to be so against the latter. In either case, third-party advertisers can attempt to frustrate XRay's auditing. We discuss each threat in turn.

Non-Malicious Web Services. Many services incorporate protections against specific automated behaviors. For example, Google makes it hard to create new accounts, although doing so remains within reach. Moreover, many services actively try to identify spammers and click fraud. Gmail includes sophisticated spam filtering mechanisms, while YouTube rate limits video viewing to prevent spam video promotion. Finally, many services rate limit access from the same IP address.

XRay-based tools must be aware of these mechanisms and scale back their activities to avoid raising red flags. For example, our prototype for Gmail, YouTube, and Amazon rate limit their output collection in the shadow accounts. Moreover, XRay's very design is sensitive to these challenges: by requiring as few accounts as possible, we minimize: (1) the load on the service imposed by auditing, and (2) the amount of input replication across shadow accounts. Moreover, XRay's workloads are often atypical of spam workloads. Our XRay Gmail plugin sends emails from one to a few other accounts, while spam is sent from one account to many other accounts.

Malicious Third-Party Advertisers. Third-party advertisers have many ways to obfuscate their targeting from XRay, particularly if it may arouse a public outcry. First, an advertiser could purposefully weaken its targeting by, for example, targeting the same ad 50% on one topic and 50% on another topic. This weakens input/output correlation and may cause XRay to infer untargeting. However, it also makes the advertisers' targeting less effective and potentially more ambiguous if their goal is to learn specific sensitive information about users. Second, an advertiser might target complex combinations of inputs that XRay's basic design cannot discover. Our accompanying technical report shows an example of how advertisers might achieve this [26]. It also extends our theoretical models so they can detect targeting on linear combinations with only a constant factor increase in the number of accounts. We plan to incorporate and evaluate these extensions in a future prototype.

Malicious Web Services. A malicious service could identify and disable shadow accounts. Identification could be based on abnormal traffic (successive reloads of email pages), data distribution within accounts (several accounts with subsets of one account), and perhaps more. XRay could be extended to add randomness and deception (e.g., fake emails, varying copies). More importantly, a collaborative approach to auditing, in which users contribute their ads and input topics in an privacy-preserving way is a promising direction for strengthening robustness against attacks. Web services cannot, after all, disable legitimate user accounts to frustrate auditing. We plan to pursue this direction in future work.

8 Discussion

XRay takes a significant step toward providing data management transparency in Web services. As an initial effort, it has a number of limitations. First, both the Set

Intersection and Bayesian algorithms assume independent targeting across accounts and over time. In reality, ad targeting is not always independent across either. For example, advertisers set daily ad budgets. When the budget runs out, an ad can stop appearing in accounts mid-experiment even though it has the targeted attributes. The system might incorrectly assume that no targeting is taking place, when it could resume the next day. XRay takes reduced coverage into account, but differences between ads can let some targeting pass unnoticed. XRay does not currently account for these dependencies, but estimating their impact is an important goal for future work.

Second, we assume that targeting noise is bounded and smaller than the targeting signal. While this condition seems to hold on the evaluated services, other services making more local decisions may be harder to audit. For example, Facebook might target ads based on friends' information, potentially creating noise that is as high as the targeting signal. A future solution might imitate the social network in shadow accounts.

Third, XRay uses Web services atypically. To the best of our knowledge, it does not violate any terms of service. It does, however, collect ads paid for by advertisers to detect correlation. Ad payment is per impression and pay per click. The former is vastly less expensive than the latter [32]. XRay creates false impressions only but never clicks on ads. A back-of-the-envelope calculation using impression pricing from [32] of $0.6/thousand impressions reveals that XRay's cost should be minimal: at most 50 cents per ad for our largest experiments.

Despite these limitations, XRay has proven itself useful for many needs, particularly in an auditing context. An auditor can craft inputs that avoid many of these limitations. For example, emails can be written to avoid as much overlap as possible and keep the size of inputs used for targeting within reasonable bounds. We hope that XRay's solid correlation components will streamline much-needed investigations – by researchers, journalists, or the FTC – into how personal data is being used.

9 Related Work

While §2.2 covered Web data protection and auditing related works, we next cover other related topics. Our work relates to recent efforts to measure various forms of personalization, such as search [21, 47], pricing [31], and ad discrimination [40]. They generally employ a methodology similar in spirit to differential correlation, but their goals differ from ours. They aim to quantify *how much* output is personalized and what *type* of information is used overall. In contrast, XRay seeks to provide fine-grained diagnosis of which *input data* generates which personalized results. Through its scaling mechanisms – unique in the personalization and data tracking literature – XRay scales well even when the relevant inputs are many and unknown in advance.

Our work also relates to a growing body of research measuring advertising networks. These networks, notably complex and difficult to crawl [3], are rendered opaque by the need to combat click fraud [9], and have been shown to be susceptible to leakage [24] and profile reconstruction attacks [6]. As for other personalization, prior studies focused mostly on macroscopic trends (e.g., What fraction of ads are targeted?) [3] or qualitative trends (e.g., Which ads are targeted toward gay males?) [20]. Various studies showed traces – but not a prevalence – of potential abuse through concealed targeting [20] and data exchange between services [46]. These works primarily focus on display advertising, and each distinguishes contextual advertising using a specific classifier with semantic categories obtained from Google's Ad Preferences Managers or another public API [28].

XRay departs significantly from these works. First, since it entirely ignores the content and even the domain of targeting, it is readily applied as-is to ads in Gmail, product recommendations, and videos. Second, while previous methods label ads as "behavioral" in bulk once other explanations fail [28], XRay remains grounded on positive evidence, and determines to *which* inputs an output should be attributed. Third, XRay's mechanisms to avoid exponential input placement and deal with overlapping inputs are unprecedented in the Web-data-tracking context. While they resemble *black box* software testing [4], the specific targeting assumption we leverage have, to our knowledge, no prior equivalent.

10 Conclusions

The tracking of personal data usage poses unique challenges. XRay shows for the first time that accurate, *fine-grained* tracking need not compromise portability and scalability. For users who care about *which* piece of their data has been targeted, it offers a unique level of precision and protection. Our work calls for and promotes the best practice of voluntary transparency, while at the same time empowering investigators and watchdogs with a significant new tool for increased vigilance.

11 Acknowledgements

We thank our shepherd, Dan Boneh, the anonymous reviewers, and numerous colleagues (Jonathan Bell, Sandra Kaplan, Michael Keller, Yoshi Kohno, Hank Levy, Yang Tang, Nicolas Viennot, and Junfeng Yang) for their valuable feedback. This work was supported by DARPA Contract FA8650-11-C-7190, NSF CNS-1351089 and CNS-1254035, Google, and Microsoft.

References

[1] Adblock plus. `https://adblockplus.org`.

[2] Amazon. Product categories. http://services.amazon.com/services/soa-approval-category.htm.

[3] P. Barford, I. Canadi, D. Krushevskaja, Q. Ma, and S. Muthukrishnan. Adscape: Harvesting and Analyzing Online Display Ads. *In Proc. of the 23nd International Conference on WWW*, 2014.

[4] B. Beizer. *Black-Box Testing*. Techniques for Functional Testing of Software and Systems. John Wiley & Sons, May 1995.

[5] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano. Public Key Encryption with Keyword Search. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, pages 506–522. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[6] C. Castelluccia, M. A. Kaafar, and M. Tran. Betrayed by your ads! *PETS'12: Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, 2012.

[7] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the 11th IEEE Symposium on Computers and Communications*, 2006.

[8] Chrome web store - collusion for chrome. https://chrome.google.com/webstore/detail/collusion-for-chrome/ganlifbpkcplnldliibcbegplfmcfigp.

[9] V. Dave, S. Guha, and Y. Zhang. Measuring and fingerprinting click-spam in ad networks. In *SIGCOMM '12: Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Crchitectures, and Protocols for Computer Communication*. ACM Request Permissions, Aug. 2012.

[10] N. Diakopoulos. Algorithmic accountability reporting: On the investigation of black boxes. Tow Center for Digital Journalism, Columbia University. February, 2014.

[11] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. *Technical Report*, 2004.

[12] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[13] M. Fredrikson and B. Livshits. RePriv: Re-imagining Content Personalization and In-browser Privacy. *2011 IEEE Symposium on Security and Privacy*, pages 131–146, 2011.

[14] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of USENIX Security*, 2009.

[15] C. Gentry. Fhe using ideal lattices. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, 2009.

[16] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *In Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[17] Google. Adsense categories. https://support.google.com/adsense/answer/3016459.

[18] J. Gould. SafeGov.org - Google admits data mining student emails in its free education apps, 2014.

[19] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[20] S. Guha, B. Cheng, and P. Francis. Challenges in measuring online advertising systems. In *IMC '10: Proceedings of the 10th Annual Conference on Internet Measurement*, 2010.

[21] A. Hannak, P. Sapiezynski, A. M. Kakhki, B. Krishnamurthy, D. Lazer, A. Mislove, and C. Wilson. Measuring personalization of web search. In *WWW '13: Proceedings of the 22nd International Conference on World Wide Web*, 2013.

[22] A. L. Hughes and L. Palen. Twitter adoption and use in mass convergence and emergency events. *International Journal of Emergency Management*, 2009.

[23] A. Korolova. Privacy Violations Using Microtargeted Ads: A Case Study. In *ICDM Workshops*, 2010.

[24] A. Korolova. Privacy violations using microtargeted ads: A case study. *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 474–482, 2010.

[25] B. Krishnamurthy and C. E. Wills. On the leakage of personally identifiable information via online social networks. In *Proc. of the 2nd ACM Workshop on Online Social Networks*, 2009.

[26] M. Lecuyer, G. Ducoffe, F. Lan, A. Papancea, T. Petsios, R. Spahn, A. Chaintreau, and R. Geambasu. XRay: Enhancing the Web's Transparency with Differential Correlation. Technical report, CS Department, Columbia University, 2014.

[27] Lightbeam. http://www.mozilla.org/lightbeam/.

[28] B. Liu, A. Sheth, U. Weinsberg, J. Chandrashekar, and R. Govindan. AdReveal: improving transparency into online targeted advertising. In *Proc. of the Twelfth ACM Workshop on Hot Topics in Networks*, 2013.

[29] D. Mattioli. WSJ.com - On Orbitz, Mac Users Steered to Pricier Hotels, 2012.

[30] V. McKalin. Techtimes.com - google: We promise not to spy on student email accounts to deliver ads, 2014.

[31] J. Mikians, L. Gyarmati, V. Erramilli, and N. Laoutaris. Detecting price and search discrimination on the internet. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 79–84, 2012.

[32] L. Olejnik, T. Minh-Dung, C. Castelluccia, et al. Selling off privacy at auction. In *In Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, 2011.

[34] F. Roesner. sharemenot.cs.washington.edu.

[35] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *NSDI'12: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Apr. 2012.

[36] A. Sadilek and H. Kautz. Modeling the impact of lifestyle on health at scale. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, 2013.

[37] SafeGov.org. Declaration of Kyle C. Wong in Support of Google Inc.'s Opposition to Plaintiffs' Motion for Class Certification, 2013.

[38] Snapchat. http://blog.snapchat.com/.

[39] Snapchat blog - how snaps are stored and deleted.

[40] L. Sweeney. Discrimination in online ad delivery. *Communications of the ACM*, 56(5), May 2013.

[41] The Guardian. Snapchat's expired snaps are not deleted, just hidden, 2014.

[42] V. Toubiana, A. Narayanan, and D. Boneh. Adnostic: Privacy preserving targeted advertising. *Proc. NDSS*, 2010.

[43] J. Valentino-Devries, J. Singer-Vine, and A. Soltani. WSJ.com - Websites Vary Prices, Deals Based on Users' Information, 2012.

[44] X. Wang, M. Gerber, and D. Brown. Automatic crime prediction using events extracted from twitter posts. In S. Yang, A. Greenberg, and M. Endsley, editors, *Social Computing, Behavioral - Cultural Modeling and Prediction*, volume 7227 of *Lecture Notes in Computer Science*, pages 231–238. 2012.

[45] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proc. of USENIX Security*, 1999.

[46] C. E. Wills and C. Tatar. Understanding What They Do with What They Know. *WPES '12: Proceedings of the 12th Annual ACM Workshop on Privacy in the Electronic Society*, 2012.

[47] X. Xing, W. Meng, D. Doozan, N. Feamster, W. Lee, and A. C. Snoeren. Exposing Inconsistent Web Search Results with Bobble. *Passive and Active Measurements Conference*, 2014.

[48] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical Report UCB/EECS-2009-145, 2009.

[49] P. R. Zimmermann. The official PGP user's guide. 1995.

A Proof of Theorem 1

A.1 Targeting functions, Axioms and Core Family

A *combination* $\mathscr{C}$ of order r, also called r_combination, is a subset of r elements among the N inputs.

Each given ad is associated with a *targeting function* defined as a mapping f from any subset $\mathscr{C}$ of the N inputs into $\{0,1\}$, where $f(\mathscr{C}) = 1$ denotes that an account containing $\mathscr{C}$ as inputs should be targeted. By convention, untargeted ads are associated with the null function $f(.) = 0$. Any targeting function f satisfies two axioms:

- **monotonicity**: $\mathscr{C} \subseteq \mathscr{C}' \implies f(\mathscr{C}) \leq f(\mathscr{C}')$.
- **input-sensitivity**: $\exists \mathscr{C}, \mathscr{C}'$ s.t. $f(\mathscr{C}) = 0, f(\mathscr{C}') = 1$.

Monotonicity simply reflects that an account with strictly more interest or hobbies should in theory be relevant to more ads, and never to less. Input sensitivity prevents the degenerate case where a targeting function is constant.

A *family* S of size l is any collection of l distinct combination. The *order* of this family is defined as the largest order of a combination it contains. For any family S, one can define a targeting function that takes value 1 whenever the subset contains at least one combination in S. Indeed, as shown in [26], the converse is true:

Lemma 1 *For each monotone, input-sensitive targeting function there exists a unique family S satisfying:*

(i) S has size l and order r and it explains *f, which means $f(\mathscr{C}) = 1$ holds if and only if $\exists \mathscr{C}' \in S, \mathscr{C}' \subseteq \mathscr{C}$.*

(ii) No family of size $l' < l$ explains f.

(iii) No family of order $r' < r$ explains f.

Hence, associated with each ad and therefore each targeting function is a unique family of input combination that are targeted, called the ad's *core family*, and we now sketch why it is correctly identified by our algorithm.

A.2 Algorithm and Correctness

For any family of subsets S and fraction $0 \leq x \leq 1$, we say a subset of inputs $\mathscr{C}$ is an x_intersecting subset of S if x subsets in S have at least one input in $\mathscr{C}$. Our proof exploits an original connection between small intersecting subsets (that can be found efficiently) to show how they can reveal a core family. One way to understand why is the following: say, for instance, that the targeting function f takes value 1 exactly when one of the inputs within $\mathscr{C}$ is found in the account. Then $\mathscr{C}$ is exactly the union of inputs found in the core family and intersects all accounts within scope, i.e., forms a large fraction of those receiving the ad.

The key property to explain our algorithm is random subsets. We can show under the conditions of the theorem that there exists $0 < x < 1$ that satisfies two properties related to the inputs of accounts receiving the ads: (1) if targeting does not occur, then with a large probability we cannot find a subset of l inputs that meets at least a fraction x of the accounts seeing the ad, and (2) if targeting does occur, we have accounts receiving the ads for various reasons, within and outside the targeting scope. But we can show with high probability that at least a fraction x of them are within scope and hence must include one combination in the core family. Since with each core family of size l one can associate an intersecting subset that contains at most l elements, checking the existence of such a subset reveals the presence of targeting.

This explains why an algorithm can qualitatively conclude whether targeting occurs or not, but it does not explain how the core family can be computed. However, leveraging stronger results of random subsets allows to apply the same rule recursively, offering multiple ways to determine exactly the core family even with a polynomial number of operations.

More formally, we define: A random *Bernoulli subset*, denoted by $B(n,p)$, is a subset such that any of n elements is contained with probability p independently of all others. A random *Bernoulli family* of size m is a collection of m independent Bernouilli subsets. We first show property (1) above more formally:

Lemma 2 *Let $x > 0$, $s \in \mathbb{N}$, $p < 1 - (1-x)^{\frac{1}{s}}$, and a Bernouilli family $B_1(n,p), B_2(n,p), \ldots, B_m(n,p)$. For any $\varepsilon > 0$ and polynomial P of degree $\leq r$, there exists $A > 0$ such that with probability $\left(1 - \frac{\varepsilon}{P(n)}\right)$ no x_intersection subset exists of size s whenever we have:*

$$m \geq A \cdot ((s+r)\ln(n) + \ln(1/\varepsilon)) .$$

To prove property (2), we need to bound, among accounts receiving an ad, the fraction that is outside the scope of targeting but still receives the ads because $p_{\text{out}} > 0$. Formally, we have:

Lemma 3 *Let $x > 0$, $\alpha > 0$, and a core family of size l and order r p_{in}, p_{out} where we have $p_{out}/p_{in} < \frac{1-x}{x}\frac{\alpha^r}{1-\alpha^r}$. Let $\mathscr{C}$ be a combination of order r.*

For any $\varepsilon > 0$ and polynomial P of degree $\leq r$, there exists $A > 0$ such that with probability $(1 - \varepsilon/P(n))$ the following holds: Among accounts containing $\mathscr{C}$ and receiving the ad, at least x fraction of them is within the targeting scope whenever we have:

$$m \geq A \cdot (r\ln(n) + \ln(1/\varepsilon)) .$$

The two lemmas above (proved in [26]) can be combined whenever α satisfies the inequality for p in the first lemma, which shows that an algorithm can detect the presence of targeting whenever $p_{\text{out}}/p_{\text{in}} < \frac{1-x}{x}\frac{(1-(1-x)^{\frac{1}{l}})^r}{1-(1-(1-x)^{\frac{1}{l}})^r}$.

A naive exponential algorithm could be used to exhaustively search for a core family using this brick. We also show that a polynomial algorithm can refine this analysis to compute the core family at the expense of a more complex recursion in [26].

An Internet-Wide View of Internet-Wide Scanning

Zakir Durumeric
University of Michigan
zakir@umich.edu

Michael Bailey
University of Michigan
mibailey@umich.edu

J. Alex Halderman
University of Michigan
jhalderm@umich.edu

Abstract

While it is widely known that port scanning is widespread, neither the scanning landscape nor the defensive reactions of network operators have been measured at Internet scale. In this work, we analyze data from a large network telescope to study scanning activity from the past year, uncovering large horizontal scan operations and identifying broad patterns in scanning behavior. We present an analysis of who is scanning, what services are being targeted, and the impact of new scanners on the overall landscape. We also analyze the scanning behavior triggered by recent vulnerabilities in Linksys routers, OpenSSL, and NTP. We empirically analyze the defensive behaviors that organizations employ against scanning, shedding light on who detects scanning behavior, which networks blacklist scanning, and how scan recipients respond to scans conducted by researchers. We conclude with recommendations for institutions performing scans and with implications of recent changes in scanning behavior for researchers and network operators.

1 Introduction

Internet-wide scanning is a powerful technique used by researchers to study and measure the Internet and by attackers to discover vulnerable hosts *en masse*. It is well known that port scanning is pervasive — including both large horizontal scans of a single port and distributed scanning from infected botnet hosts [5, 14, 15, 28, 39, 45]. However, the past year saw the introduction of two high-speed scanning tools, ZMap [19] and Masscan [23], which have shifted the scanning landscape by reducing the time to scan the IPv4 address space from months to minutes.

In this study, we examine the practice of Internet-wide scanning and explore the impact of these radically faster tools using measurement data from a large network telescope [13, 37, 46]. We analyze scan traffic from the past year, develop heuristics for recognizing large horizontal scanning, and successfully fingerprint ZMap and Masscan. We present a broad view of the current scanning landscape, including analyzing who is performing large scans, what protocols they target, and what software and providers they use. In some cases we can determine the identity of the scanners and the intent of their scans.

We find that scanning practice has changed dramatically since previous studies from 5–10 years ago [5,39,45]. Many large, likely malicious scans now originate from bullet-proof hosting providers instead of from botnets. Internet-scale horizontal scans have become common. Almost 80% of non-Conficker probe traffic originates from scans targeting $\geq$1% of the IPv4 address space and 68% from scans targeting $\geq$10%.

To understand how and why people are conducting scans, we attempt to identify individual large-scale scanning operations. We find that researchers are utilizing new scanning tools such as ZMap to cull DDoS attacks and measure distributed systems, but we also uncover evidence that attackers are using these tools to quickly find vulnerable hosts. In three case studies, we investigate scanning behavior following the disclosure of the OpenSSL Heartbleed vulnerability [36], vulnerabilities in Linksys routers, and vulnerabilities in NTP servers. In each instance, the vast majority of probe traffic originated from large, single-origin scanners. For the Linksys and OpenSSL vulnerabilities, we observed attackers applying ZMap from international bullet-proof hosting providers to complete full scans of the IPv4 address space within 24 hours of public vulnerability disclosure.

We also investigate the defensive mechanisms employed by network operators to detect and respond to scanning. Even in the most favorable case for detection — when repeated, aggressive scan traffic originates from a single IP address and would be trivial to fingerprint — we find that only a minuscule fraction of organizations respond by blocking the probes. When probes are blocked, it is often after operators inadvertently find evidence of scanning during other maintenance, rather than through

automated detection. This may indicate that the vast majority of network operators do not regard scanning as a significant threat. It also validates many recently published research studies based on Internet-wide scanning, as dropped traffic and exclusion requests appear to have minimal impact on study results.

Our findings illustrate that Internet-wide scanning is a rapidly proliferating methodology among both researchers and malicious actors. Maintaining its enormous utility for defensive security research while simultaneously protecting networks from attack is a difficult challenge. Network operators need to be aware that large vulnerability scans are taking place within hours of disclosure, but they should remember that blindly blocking all networks responsible for scanning may adversely impact defensive research. Future work is needed to develop mechanisms for differentiating between benign and malicious scans. In the mean time, we recommend close cooperation between researchers and network operators.

2 Previous Work

Most similar to our work is a study in 2004 by Pang et al. [39], who performed one of the first comprehensive analyses of Internet background radiation. Their study covers many aspects of background traffic, including the most frequently scanned protocols. However, the scanning landscape has changed drastically in the last decade—the Conficker worm [40], a major source of probe traffic, appeared in 2008, and ZMap [19] and Masscan [23] were released in 2013.

In 2007, Allman et al. [5] briefly described historical trends in scan activity between 1994 and 2006. Wustrow et al. [45] again studied Internet background radiation in 2010. They noted an increase in scan traffic destined for SSH (TCP/22) and telnet (TCP/23) in 2007, as well as increased scanning activity targeting port 445 (SMB over IP) in 2009 due to Conficker. We note a different set of targeted services and other changes in scanning dynamics since that time. Czyz et al. [14] explored background radiation in the IPv6 address space. Their work briefly touches on the presence of ICMPv6 probe traffic, but otherwise does not investigate scanning activity; we focus on the IPv4 address space.

There exists a large body of work that focuses on detecting distributed botnet scanning [22, 24, 29, 31, 43]. However, barring few exceptions, this phenomenon has remained largely hypothetical. In one exception, Javid and Paxson [28] unearthed slow but persistent SSH brute-force attacks in 2013. Similarly, Dainotti et al. analyzed distributed botnet scanning in 2011.

Real-world responses to horizontal scanning have not been previously studied. We briefly discussed reactions to our own scanning in prior work [19], but we perform a more in-depth analysis now. Leonard et al. [32] similarly describe the complaints they received when attempting to build an Internet scanner; however, our analysis is based on a much larger data set. In addition, we perform experiments to detect instances where networks block scan probes without notice.

The dynamics of performing studies on IPv4 darknet traffic have been formally documented by both Moore et al. [37] and Cooke et al. [13]. We utilize both studies when performing calculations in this work.

3 State of Scanning

In order to understand current scanning behavior, we analyzed traffic received by a large darknet over a 16-month period. We find that large-scale horizontal scanning—the process of scanning a large number of hosts on a single port—is pervasive and that, excluding Conficker, almost 80% of scan traffic originates from large scans targeting >1% of the IPv4 address space. We find evidence that many scans are being conducted by academic researchers. However, a large portion of all scanning targets services associated with vulnerabilities (e.g. Microsoft RDP, SQL Server), and the majority of scanning is completed from bullet-proof hosting providers or from China. In this section, we describe the dynamics of these scans, including identifying the services targeted, the sources of the scans, and the largest scanning operations.

3.1 Dataset and Methodology

Our dataset consists of all traffic received by a darknet operated at Merit Network for the period from January 1, 2013 to May 1, 2014. The darknet is composed of 5.5 million addresses, 0.145% of the public IPv4 address space. During this period, the darknet received an average of 1.4 billion packets, or 55 GB of traffic, per day. For non-temporal analyses, we focus on January 2014.

In order to distinguish scanning from other background traffic, we define a scan to be an instance where a source address contacted at least 100 unique addresses in our darknet (.0018% of the public IPv4 address space) on the same port and protocol at a minimum estimated Internet-wide scan rate of 10 packets per second (pps). In the case of TCP, we consider only SYN packets.

While we cannot know for sure whether a particular scan covers the entire IPv4 address space, the darknet does not respond to any incoming packets, and the majority of its parent /8 does not host any services. As such, we expect that hosts that send repeated probes to the darknet are scanning naïvely and are likely targeting a large portion of the address space.

Detecting scans Assuming a random uniform distribution of targets, the probability that a single probe packet will be detected can be modeled by a geometric distribution and the number of packets observed by our darknet modeled by a binomial distribution [37]. A scanner probing random IPv4 addresses at the slowest rate we try to detect (10 pps) will appear in our darknet with 99% confidence within 311 seconds and with 99.9% confidence within 467 seconds. We estimate the number of packets sent to the entire IPv4 address space by approximating the binomial distribution with a normal distribution.

We process the darknet traffic using libpcap [27] and apply a single-pass algorithm to identify scans. We expire scans that do not send any packets in more than 480 seconds and record scans that reach at least 100 darknet addresses before expiring. We combine scans originating from sequential addresses in a routed block, as ZMap allows users to scan from a block of addresses. We perform geolocation using the MaxMind GeoIP dataset [35].

Fingerprinting scanners We investigate open-source scanners and fingerprint the probes generated by ZMap [19] and Masscan [23]. In ZMap, the IP identification field is statically set to 54321. In Masscan, probes can be fingerprinted using the following relationship:

$$\text{ip\_id} = \text{dst\_addr} \oplus \text{dst\_port} \oplus \text{tcp\_seqnum}$$

Because the IP ID field is only 16 bits and has a non-negligible chance of randomly being either of these values, we only consider scans in which all packets match one of the fingerprints. We find no easily identifiable characteristics for Nmap [33] probes.

3.2 Scan Dynamics

We detected 10.8 million scans from 1.76 million hosts during January 2014. Of these, 4.5 million (41.7%) are TCP SYN scans targeting less than 1% of the IPv4 address space on port 445 and are likely attributable to the Conficker worm [40]. Excluding Conficker traffic, the scans are composed of 56.4% TCP SYN packets, 35.0% UDP packets, and 8.6% ICMP echo request packets. Only 17,918 scans (0.28%) targeted more than 1% of the address space, 2,699 (0.04%) targeted more than 10%, and 614 (0.01%) targeted more than 50% (see Figure 5). However, after excluding Conficker traffic, we note that 78% of probe traffic is generated by scans targeting ≥1% of the IPv4 address space, 62% by scans targeting ≥10%, and 30% by scans targeting ≥50% (see Figure 4). In other words, while there is a relatively small number of large scans (0.28%), nearly 80% of scan traffic is generated by these scans.

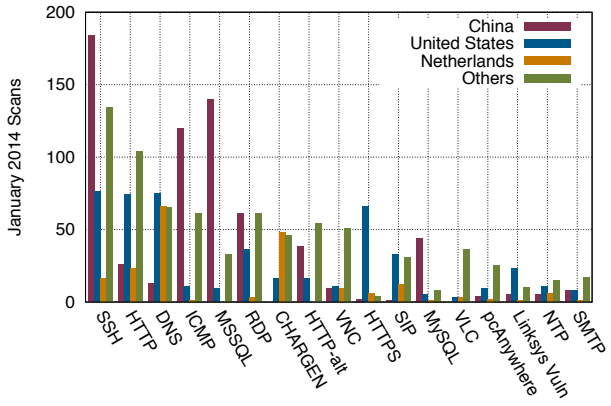


Figure 1: **Large scans (≥10%) by origin country** — Many countries have distinct scanning profiles. For example, the vast majority of MSSQL scanning takes place in China.

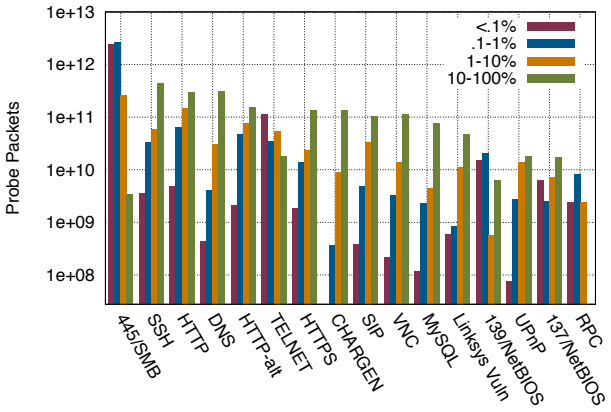


Figure 2: **Targeted ports by scan size** — Small scans target different protocols than large scans. For example, the bulk of port 445 scanning occurs in small scans, whereas port 22 is targeted by larger scans.

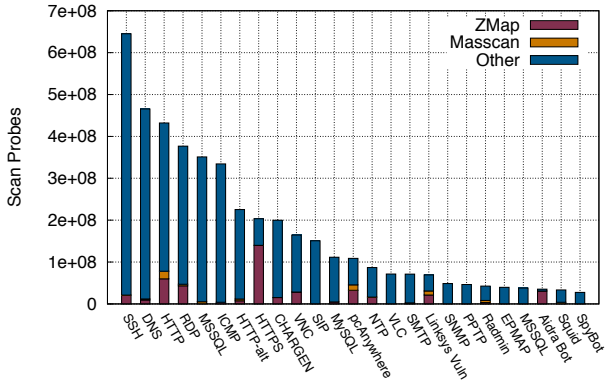


Figure 3: **Large scans (≥10%) by software** — We fingerprint ZMap and Masscan probes and present the breakdown of large scans that use these scanners.

3.3 Targeted Services

Close to half of all scan traffic (48.9%) targets NetBIOS (TCP/445)—5.4 trillion SYN probes in January 2014 alone. Of these packets, 95.1% originate from small scans—scans targeting <10% of the IPv4 address space—and are likely attributable to Conficker [40, 45]. We note that small scans show different characteristics than large scans. For example, while SSH is the most targeted service in large scans, it is the seventh most targeted in small scans, accounting for only 1.3% of scan traffic.

For the most part, the protocols being targeted are not surprising, although they have shifted from previous studies in 2004 and 2010—we show the differences in Table 3. In both large and small scans, there appear to be a mix of protocols frequently associated with vulnerability scanning (e.g. Microsoft RDP, telnet, Microsoft SQL server, and VNC) as well protocols frequently studied by academic researchers (e.g. HTTP, HTTPS, SSH). We show the differences in Figure 2 and the breakdown of frequently targeted services in Tables 1 and 2.

Despite the fact that most scans originate from large international hosting providers, countries display differences in targeted protocols—particularly China, which performs regular scans against SSH, SQL Server, and Microsoft RDP. For example, while Microsoft Remote Desktop Protocol (RDP) is the fourth most scanned protocol, 77% of scans and 76% of probe packets originate from China. The second most active country (United States) is responsible for only 5.4% of probe traffic. A similar pattern emerges for ICMP echo request scans, MySQL and SSH. We show the differences by country for the top ports in Figure 1.

3.4 Scan Sources

While large scans originate from 68 countries, 76% of scan traffic originates from only five countries: China, the United States, Germany, the Netherlands, and Russia. We list the top countries that performed horizontal scans in Table 4 and the CDF in Figure 7.

While the United States and China have large allocations of address space, Germany and the Netherlands do not. In order to understand why a disproportionate amount of scan traffic is originating from smaller countries, we consider the ASes from which scans are being completed. We find that scans targeting ≥10% of the IPv4 address space occur from only 350 ASes (Figure 8). We manually classify the top 100 ASes, finding that 49 are dedicated hosting services or collocation centers, 31 are Internet service providers, 4 are academic institutions, 3 are corporations, and 13 are unidentifiable networks in China.

In the case of the Netherlands, 93% of probe traffic originates from five hosting providers: Ecatel Network,

2004 [39]	2010 [45]	2014
HTTP (80)	SMB-IP (445)	SMB-IP (445)
NetBIOS (135)	NetBIOS (139)	ICMP Ping
NetBIOS (139)	eMule (4662)	SSH (20)
DameWare (6129)	HTTP (80)	HTTP (80)
MyDoom (3127)	NetBIOS (135)	RDP (3389)

Table 3: **Temporal differences in targeted protocols** — Previous studies on background radiation show a distinct set of most frequently targeted services.

Country	Scans	Country	Scans
China	805 (31%)	Poland	61 (2.3%)
United States	582 (22%)	Korea	61 (2.3%)
Germany	247 (9.5%)	Ukraine	43 (1.7%)
Netherlands	229 (8.8%)	Brazil	34 (1.3%)
Russia	127 (4.8%)	Other	337 (13%)
France	81 (3.1%)		

Table 4: **Large scans (≥10%) by country** — A small number of countries are responsible for the majority of large scans.

Ecatel Network (NL)	Thor Data Center (IS)
Plus Server (DE)	Psychz Networks (US)
Slask Data Center (PL)	ServerStack, Inc. (US)
SingleHop (US)	Amazon.com, Inc. (US)
CariNet, Inc. (US)	LeaseWeb (NL)
SERVER4YOU (DE)	Digital Ocean, Inc. (US)
OVH Systems (UK)	GorillaServers, Inc. (US)

Table 5: **Top providers originating scan traffic** — The majority of scan probes came from large dedicated hosting and colocation providers.

Contact Point	Organizations	
Email listed on website	108	(59.7%)
WHOIS abuse contact	31	(17.1%)
Security office	22	(12.2%)
Specific individuals (e.g. CSO, CIO)	9	(5.0%)
Departmental helpdesk	5	(2.8%)
Other email contacts (e.g. postmaster)	6	(3.3%)
IT help desk phone	2	(1.1%)

Table 6: **Exclusion point of contact** — We track how organizations contacted our research team to request exclusion from future scans.

SMB over IP (TCP/445)	71.8%	SIP (UDP/5060)	0.5%	NetBIOS Helper (TCP/49153)	0.2%
ICMP Echo Request	4.8%	NetBIOS Session (TCP/139)	0.5%	Linksys Vuln. (TCP/32764)	0.2%
Microsoft RDP (TCP/3389)	3.1%	DNS (UDP/53)	0.5%	ASF-RMCP (UDP/623)	0.1%
HTTP (TCP/80)	3.0%	VLC (UDP/1234)	0.4%	SNMP (UDP/161)	0.1%
Telnet (TCP/23)	2.8%	SMTP (TCP/25)	0.2%	CHARGEN (UDP/19)	0.1%
Alt-HTTP (TCP/8080)	1.7%	VNC (TCP/5900)	0.2%	MongoDB (TCP/27017)	0.1%
SSH (TCP/22)	1.3%	Microsoft SSDP (UDP/1900)	0.2%	pcAnywhere (UDP/5632)	0.1%
HTTPS (TCP/443)	0.5%	NetBIOS Name Svc (TCP/137)	0.2%	Other	7.4%

Table 1: **Commonly targeted services for small scans (targeting <10% of the IPv4 address space)**

SSH (TCP/22)	12.5%	CHARGEN (UDP/19)	3.9%	Linksys Vuln. (TCP/32764)	1.3%
DNS (UDP/53)	9.0%	VNC (TCP/5900)	3.2%	SNMP (UDP/161)	1.0%
HTTP (TCP/80)	8.4%	SIP (UDP/5060)	2.9%	Micorosft PPTP (TCP/1723)	0.9%
Microsoft RDP (TCP/3389)	7.3%	MySQL (TCP/3306)	2.2%	Radmin (TCP/4899)	0.8%
SQL Server (TCP/1433)	6.9%	pcAnywhere (TCP/5631)	2.1%	DCOM SCM (TCP/UDP/135)	0.8%
ICMP Echo Request	6.5%	NTP (UDP/123)	1.7%	MS SQL Server (UDP/1434)	0.7%
Alt-HTTP (TCP/8080)	4.4%	VLC (UDP/1234)	1.4%	Aidra Botnet (TCP/4028)	0.7%
HTTPS (TCP/443)	4.0%	SMTP (TCP/25)	1.4%	Other	16.2%

Table 2: **Commonly targeted services for large scans (targeting $\geq$10% of the IPv4 address space)**

LeaseWeb, WorldStream, Datacenter, Nedzone, and TransIP. We note that Ecatel was one of the hosting providers that Hurricane Electric stopped peering with in 2008 due to spam traffic and malware hosting [12]. In Germany, PlusServer was responsible for 45% of probe traffic. In the United States, scanning was present from 440 ASes, but a small handful of hosting providers were responsible for 39% of scan traffic[1]. We list the hosting providers and collocation centers responsible for the most scan traffic in Table 5.

3.5 Regularly Scheduled Scans

We investigate the 25 most aggressive scanners and find several examples of both academic research scans and likely malicious groups performing repeated scans. In many of the cases where scans were performed from an academic network, researchers provided information on the purpose of their scanning. However, most scans take place from bullet-proof hosting providers or from China and provide no identifying information.

The academic and non-profit scans primarily focus on protocols used for DDoS amplification and studying cryptographic ecosystems (e.g. HTTPS and SSH). All of the groups we identified explained the purpose of their scanning and allow operators to request exclusion. Similarly, several security companies also completed scans. The Shodan Search Engine [34] was the only security group that we were able to detect that did not provide information over the web on scan addresses.

The University of Michigan performs regular ZMap scans for HTTPS hosts in order to track the certificate authority ecosystem [18, 19, 25, 47]; their data is available online at https://scans.io [17]. Ruhr-Universität Bochum completes weekly scans on ports 53, 80, 123, 137, 161, and 1900 in order to measure amplification attacks [42]. The Shadow Server Open Resolver Scanning Project [4] performs daily scans for DNS servers (UDP/53); their scanning machines are hosted by AOL. One of their hosts generated the most probes of any source in our sample — an estimated 97 billion packets in January 2014 alone. Similarly, the Open Resolver Project [3] completes weekly scans for DNS (UDP/53) and NTP (UDP/123) servers. All these institutions provide information on scan intent and how to request exclusion on a simple website at the scan source IPs.

Shodan completed 2,294 scans targeting 53 ports, sending an estimated 209 billion probes from six servers[2] in January 2014. The scans most frequently targeted ports 443, 80, 53, 32764, 1900, 23, 623, 27017, 161, and 137. Errata Security executed 89 scans of common ports using their Masscan tool. Rapid7 performed 13 scans of common ports using ZMap; their datasets are publicly available at https://scans.io [17].

There are two daily ICMP echo request scans from Guangzhou, China that jointly target an average estimated 77% of the IPv4 address space[3]. The hosts only appear to be used for these ICMP scans. A second host in

[1] CariNet (13.0%), SingleHop (11.4%), Hosting Solutions International (4.37%), Versaweb, LLC (3.46%), Psychz Networks (2.2%), Amazon.com (2.1%), and Leaseweb USA (2.0%)

[2] 198.20.69.98, 198.20.69.74, 198.20.70.114, 66.240.192.138, 71.6.5.200, and 71.6.167.142

[3] 113.108.2.117, 159.253.146.141, 220.177.198.034, and 59.46.161.130

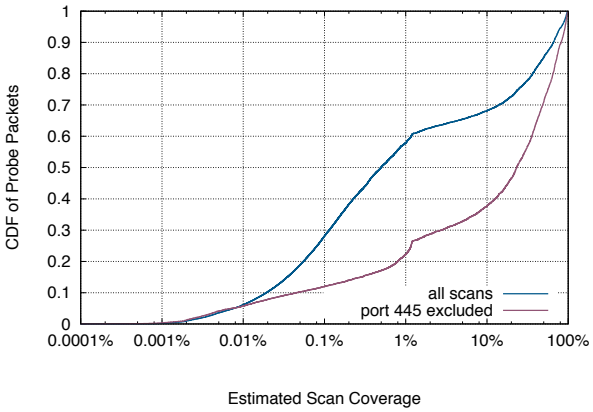


Figure 4: **CDF of scan traffic** — 40% of probes originated from scans targeting ≥1% of the IPv4 space and 30% from scans targeting ≥10%.

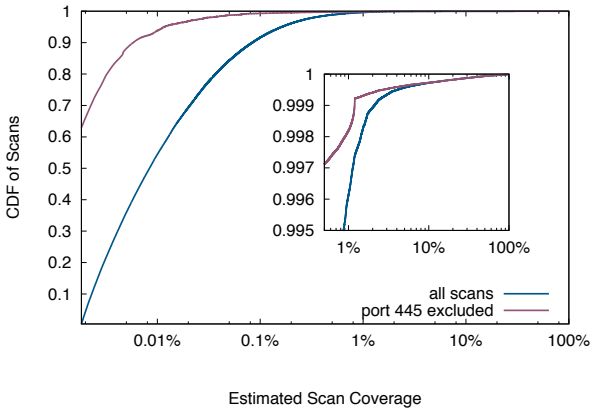


Figure 5: **CDF of scan coverage** — 45.5% of scans achieved 0.01% coverage, 8.37% achieved 0.1% coverage, and 0.38% achieved ≥1% coverage.

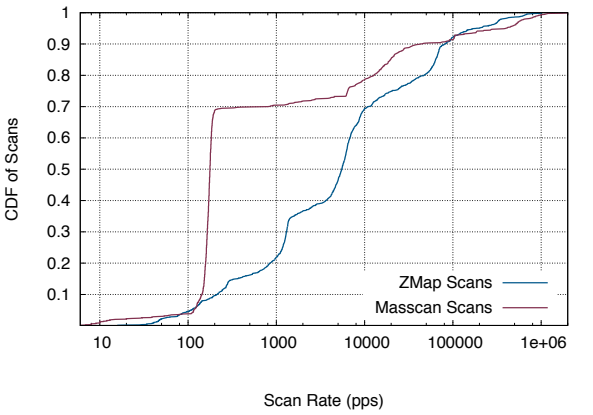


Figure 6: **CDF of scan rate** — The fastest scans operated at 2.2 Mpps (about 1.5 Gbps). However, less than 10% of scans exceeded 100 Mbps.

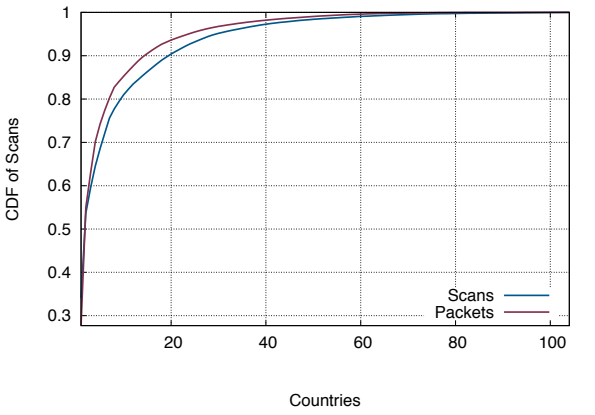


Figure 7: **CDF of scanning countries** — 76% of scans targeting ≥10% of the IPv4 address space originated from only five countries.

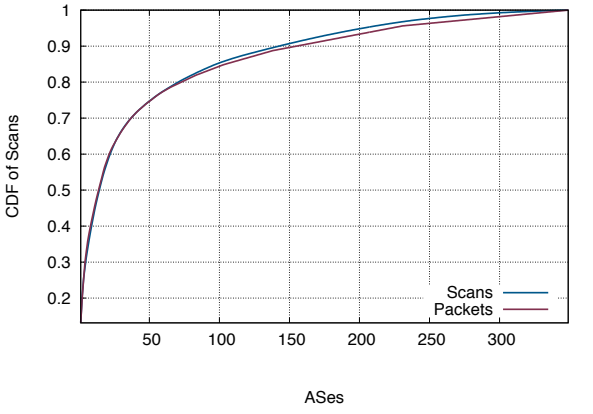


Figure 8: **CDF of scanning ASes** — Scans targeting ≥10% of the address space originated from only 350 ASes, many of them large hosting providers.

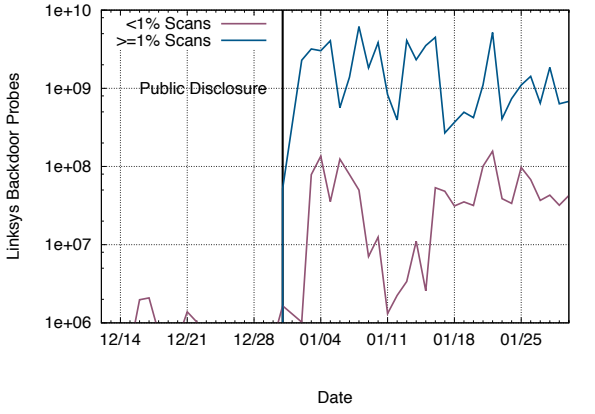


Figure 9: **Linksys backdoor** — Widespread scanning of port 32764 started within hours of the disclosure of a Linksys backdoor on that port.

Guangzhou (113.108.21.16) performs regular daily SYN scans of TCP/0, and a host in Changzhi (218.26.89.179) performs similar scans targeting SSH (TCP/22). We note that while TCP/0 is reserved, it is frequently used for fingerprinting network stacks and because it is not possible to block the port on some firewalls.

The remaining hosts in the top 25 most active scanners repeatedly scanned well-known ports and were hosted from large hosting providers in Germany, Iceland, Romania, Poland, Russia, and China. None of the hosts provided any identifying information in WHOIS records, reverse DNS records, or websites.

3.6 ZMap and Masscan Usage

The majority of scans targeting $\geq$10% of the IPv4 address space used neither ZMap nor Masscan. However, as scan coverage increases, the probability that a scanner uses ZMap steeply increases. ZMap was utilized for 133 (21.7%) of the 614 scans of more than 50% of the IPv4 address space in January 2014; Masscan was used for 21 (3.4%). Of the 242 ZMap scans targeting $\geq$10% of the address space, 70 (30%) targeted HTTP (TCP/80) and HTTPS (TCP/443) and were conducted by academic institutions and other clearly identifiable researchers. We show a breakdown of what scans used various scanners in Figure 3.

3.7 Estimated Scan Rate

In order to estimate the resources that scanners have available, we consider the estimated scan rate observed from ZMap and Masscan scans. We choose to utilize these as our metric for scan rate because the randomization algorithms are approximately uniformly random. We find that hosts are scanning between 13 pps and 1.02 million pps using ZMap and between 5 pps and 2.2 million pps — slightly more than 1.5 Gbps — using Masscan. While both tools support scanning at over 1 Gbps, all but a handful of scans were operated at much lower speeds. As shown in Figure 6, more than 90% of scans operate at under 100 Mbps, and over 70% are operated at under 10 Mbps.

4 Case Studies

Recent advances in high-speed scanning have altered the security landscape, making it possible for attackers to complete large-scale scans for vulnerable hosts within hours of a vulnerability's disclosure. In this section, we analyze scanning related to three recent vulnerabilities that affected Linksys routers, OpenSSL, and NTP servers. We find that likely attackers are taking advantage of new tools: they have started to use ZMap and Masscan from bullet-proof hosting providers instead of using distributed botnet scans. In the cases of the Linksys backdoor and the Heartbleed vulnerability, attackers began scans within 48 hours of public disclosure. We note that while conducting single-origin scans from bullet-proof hosting providers may lower the burden for attackers, it may also allow defenders to more easily detect and block scanning activity and identify the malicious actors.

4.1 Linksys Backdoor

In late December 2013, Eloi Vanderbeken disclosed a backdoor in common Cisco, Linksys, and Netgear home and small business routers [44]. The backdoor allowed full, unauthenticated, remote access to routers over an undocumented ephemeral port, TCP/32764. While there was previously only negligible traffic to the port, traffic spiked on January 2, 2014 when news sources began to cover the story [1,11,21]. There remained an average, sustained 1.98 billion estimated probe packets and 99.55 GB of traffic per day through the end of January (Figure 9).

After the disclosure, 22 hosts completed 43 scans targeting port 32764 on $\geq$1% of the IPv4 address space. Shodan [34] started scanning on December 31, 2013, within 48 hours of the disclosure, and continued to scan throughout January, approximately daily. Within one week, security consulting groups began scanning: Errata Security on January 7, M5 Computer Security on January 13, and Rapid7 on January 22. Two academic institutions, Katholieke Universiteit Leuven and Naukowa i Akademicka Sieć Komputerowa completed scans on January 3 and 6, respectively. Between January 14–16, two Chinese hosts (AS4808/China169 Beijing Province Network) completed scans. The remaining scans were performed from dedicated hosting providers[4]. No identifying information was found on any of the scanning hosts.

All non-Shodan scans utilized ZMap (71%) or Masscan (29%). Surprisingly, 98% of the probes targeting port 32764 were part of large scans targeting $\geq$1% of the IPv4 space, and 79% of probes were part of scans targeting $\geq$10%. In other words, scan traffic was not from a large number of distributed botnets hosts, but rather a small number of high-speed scanners.

While we cannot definitively determine the intent of the hosts in colocation centers, several of the providers have reputations for hosting malware and spammers, and for turning a blind eye to malicious behavior [12]. Assuming that customers of these providers are malicious, this implies that attackers completed comprehensive scans within 48 hours of disclosure using ZMap and Masscan from bullet-proof hosting providers.

[4] Hetzner Online AG (DE), UrDN/Ukranian Data Network (Ukraine), Ecatel Network (NL), Kyiv Optic Networks (Ukraine), root (Luxembourg), Digital Ocean, (US), Cyberdyne (Sweden), and Enzu (US)

4.2 Heartbleed Vulnerability

The Heartbleed Bug is a vulnerability in the OpenSSL cryptographic library [7] that was discovered in March 2014 and publicly disclosed on April 7, 2014 [36]. The vulnerability allows attackers to remotely dump arbitrary private data (e.g. cryptographic keys, usernames, and passwords) from the memory of many popular servers that support TLS, including Apache HTTP Server and nginx [36].

In the week following the disclosure, we detected 53 scans from 27 hosts targeting HTTPS. In comparison, in the week prior to the disclosure, there were 29 scans from 16 hosts. Unlike the Linksys vulnerability, there was not a sustained increase in scanning behavior. However, scan traffic was temporarily more than doubled for several days following the public disclosure.

While we do not know whether the scanners intended to exploit the vulnerability, we can detect which hosts began scanning for the first time following the disclosure. Of the 29 HTTPS scans seen prior to the disclosure, seven were daily scans from the University of Michigan, one was executed as part of Rapid7's SSL Sonar Project, and one belonged to the Shodan Project. A Chinese host (218.77.79.34) also performed daily scans. The remaining scans were operated out of bullet-proof hosting providers in the US, Great Britain, Poland, France, Iceland, and the Netherlands; none of them provided any identifying information.

Only 5 of the 27 hosts found scanning after the disclosure had previously been seen scanning on port 443, and only 3 had performed any scanning in 2014. The only recognizable organizations scanning in the week following the disclosure were the University of Michigan, Technische Universitaet Muenchen, Rapid7, Errata Security, and Nagravision. The remainder of the scans were completed from China and bullet-proof hosting providers. Within 24 hours of the vulnerability release, scanning began from China — 20 of the 53 scans (38%) originated from China. The remaining scans occurred from Rackspace, Cyberdyne, SingleHop, CariNet, Ecatel, myLoc, and Amazon EC2. 74% of the scans used ZMap; 21% used Masscan. Only three scans (6%) used other software.

4.3 NTP DDoS Attacks

Network Time Protocol (UDP/123) is a protocol that allows servers to synchronize time. In December 2013, attackers began to use NTP to perform denial-of-service amplification, in a similar way to how DNS had been abused in the past. Traffic from NTP servers began to rise around December 8, 2013 [2] and in February 2014, attackers attempted to DDoS a Cloudflare customer with over 400 Gbps of NTP traffic — one of the largest ever DDoS attacks [41].

The scanning behavior surrounding NTP is similar to what we observed for the Linksys backdoor and the Heartbleed vulnerability. Specifically, 97.3% of probe traffic destined for NTP was part of large scans (targeting $>1\%$), rather than from distributed botnet scanning. In January 2014, 29 scans from 19 hosts targeted NTP (UDP/123); 8 of the hosts used ZMap; 1 used Masscan. Three groups completed regular scans: Ruhr-Universitaet Bochum completed weekly scans, Shodan performed daily scans, and Errata Security completed one scan.

Three hosts in China completed full scans. The remaining 14 scans occurred from otherwise anonymous hosts in several hosting providers, including Ecatel, OVH Systems, FastReturn, Continuum Data Centers, and ONLINE S.A.S. One of the IPs hosts a website for the "Openbomb Drone Project" and also hosts the website http://ra.pe; the scan from the host only achieved 3% coverage; another one of the IPs hosts a site stating "#yolo"; one server had a reverse PTR record of "lulz".

As with the other vulnerabilities, there is no way to ascertain the intent of the scanners with certainty. However, the names and sites hosted on the IPs do not instill confidence that the hosts are maintained by responsible researchers rather than attackers.

5 Defensive Measures

In the previous two sections, we showed that Internet-wide scanning is widespread and that likely-attackers are scanning for vulnerabilities within 48 hours of disclosure. However, it is equally important to consider the reactions and defenses of those being scanned. Not only does this help us understand the defensive ecosystem, it also provides important data to calibrate the results from scanning research. In this section, we analyze networks' reactions to scanning, including which networks detect scan activity, drop traffic from repeat scanners, and report perceived network misuse.

Despite the fact that a large number of scans are occurring from unique source IPs and could be easily detected and blocked by network intrusion detection systems, we find that only a minuscule number of organizations block scan traffic or request exclusion. Our scan subnet at the University of Michigan is responsible for the third most aggressive scanning campaign on the Internet, yet we find that only 0.05% of the IP space is inaccessible to it. Similarly, only 208 organizations have requested that we exclude their networks from our scans, reducing the IPv4 address space for study by only 0.15%.

We further uncover evidence that networks are not detecting scans proactively, but are instead stumbling upon scans after *years* of consistent scanning — most likely during other troubleshooting or maintenance. While this lack of attention paints a dismal picture of current defensive

measures, the lack of blocking and exclusion also validates many of the recent research studies that utilize active Internet-wide measurements [8–10, 16, 18–20, 25, 26, 30, 38, 42, 47], as blacklisting does not appear to significantly bias scan results.

5.1 Detecting Blocked Traffic

In order to detect networks that are dropping scan traffic, we completed simultaneous ZMap scans from our scan subnet at the University of Michigan (141.212.121.0/24) and from a subnet that had never previously been used for scanning at the Georgia Institute of Technology. These scans took place on Wednesday, February 5, 2014 between 1:00 PM EST and 23:20 EST.

While our subnet at Michigan is used for multiple ongoing scanning effort, it has primarily been used for scanning the HTTPS ecosystem [18]. Between April 2012 and February 2014, we completed 390 scans on port 443 (HTTPS). The Michigan subnet was responsible for the third most scan traffic in January 2014. The scanning hosts all have corresponding DNS PTR records, WHOIS entries, and a simple website that describes our scanning, the data we collect, recent publications, and how to request exclusion from future research scans [19]. Despite these steps, we expected that some fraction of networks had detected our scanning and opted to silently drop traffic from our subnet.

For the simultaneous scans, we chose to scan port 443 at 100,000 pps in order to compare against our historical data on HTTPS. Both hosts used Ubuntu 12.04 and ZMap 1.2.0, and both had access to a full 1 Gbps of upstream bandwidth. We performed the two scans using ZMap, selecting identical randomization seeds such that the probes from both subnets arrive at approximately the same time.

There exists the likely possibility that some hosts were lost due to random packet drop and not intentional blocking—previous measurements on our network have shown a packet loss rate of approximately 3% [19]. In order to ensure that missing hosts are inaccessible due to blacklisting and not dropped packets, we immediately completed a secondary scan from the Michigan subnet, sending three SYN probes to each missing host, and removing hosts that were missed due to random packet drop. Previous work shows that sending three packets achieves a 99.4% success-rate [19].

We analyzed the set of hosts that appeared in scans from the "clean" subnet at Georgia Tech but not in scans from the "dirty" subnet at Michigan. We aggregate inaccessible hosts by routed block and find that there are two categories of missing hosts: (1) entire routed blocks that drop all traffic and (2) sporadic hosts and small networks belonging to large ISPs that are generally unidentifiable.

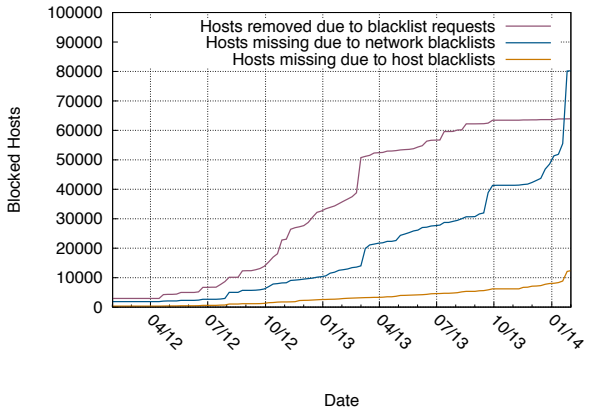


Figure 10: **Impact of blacklisting on HTTPS results** — The impact of external blacklisting and requests to be excluded from scans continue to grow over time rather than plateau.

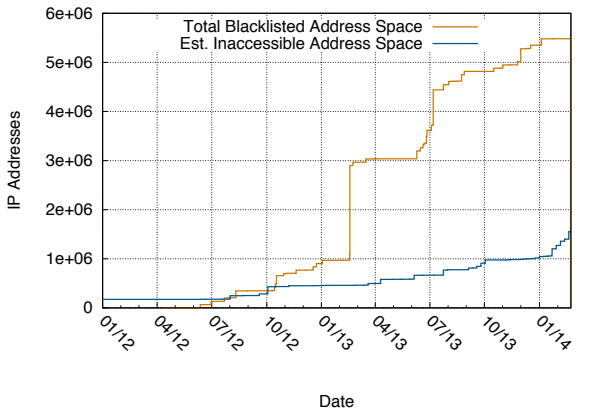


Figure 11: **Estimated impact** — We estimate inaccessible address space based on the total size of inaccessible routed blocks.

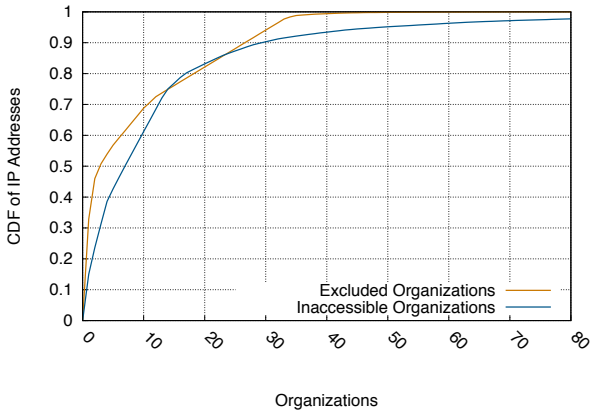


Figure 12: **CDF of blacklisting organizations** — 60% of inaccessible IPs were due to only 10 organizations.

We consider any routed block with more than three hosts in the clean subnet's scan and zero responses from the dirty subnet's scan to have blocked traffic. We find that 99,484 hosts from 612 routed blocks, 198 ASes, and 194 organizations belong to first category; 67,687 hosts belong to the second.

However, these numbers do not represent the total address space that is inaccessible to the dirty subnet, but rather the difference in hosts that respond on port 443. In order to estimate the total inaccessible address space, we consider the size of the routed blocks that appear to drop all traffic and find that these routed blocks comprise a total of 1.55 million addresses. In aggregate with the individual addresses that dropped scan traffic, we find a total of 1.62 million addresses (0.05% of the public IPv4 address space) are no longer accessible. We note that this is a lower bound of inaccessible address space as many of the individual IP addresses that we were unable to classify may represent larger, inaccessible networks. However, ultimately, only a minuscule number of organizations are detecting and blocking scan traffic.

It is important to consider not just the raw number of hosts that are inaccessible, but also the impact on the research that was being conducted by Internet-scale scanning — in our case, what percentage of the HTTPS ecosystem we are unable to measure. We compare the number of unavailable hosts to the most recent results in our HTTPS dataset, which contained TLS handshakes with 27.9 million hosts. The 167,171 inaccessible hosts would have resulted in a 0.4%–0.6% change in the result set, depending on the number of unavailable hosts that successfully completed a TLS handshake.

5.2 Organizations Blocking Scan Traffic

We identify and categorize the organizations that own each of the inaccessible routed blocks (Table 7). We note that this categorization is skewed towards organizations that are large enough to control an entire AS. Unfortunately, when attempting to classify individual IPs that blacklisted addresses, we find that most do not expose any identifying information.

As shown in Figures 10 and 11, the removal of a small number organizations resulted in large changes in the aggregate inaccessible address space — only ten organizations[5] are responsible for 60% of dropped traffic (Figure 12).

We note a bias in the countries that have blocked traffic, which we show in Table 10. However, we note that when considering the percentage of blacklisted addresses per

[5]Enzu, Corespace, Internode, Fidelity National Information Services, AR Telecom, Western Australia Department of Finance, State of Tennessee, Hershey Chocolate & Confectionery Corporation, DFN (German National Research and Education Network), and Research Organization of Information and Systems National Institute of Informatics (Japan)

Type	Organizations	Hosts
Internet service provider	73	389,120
Corporation	36	448,000
Hosting provider	34	344,832
Government	22	299,008
Academic institution	12	255,232
Small/medium business	12	63,232
Unknown	6	1,792
Total	195	1,801,216

Table 7: **Organizations that filter scans** — We categorize the organizations that blacklist scan traffic.

Type	Organizations	Hosts
Small/medium business	45	391,358
Individual	39	102
Corporation	30	671,060
Academic institution	19	1,654,401
Government	13	926,210
Internet service provider	6	1,838,827
Unknown	5	32,772
Total	157	5,514,730

Table 8: **Organizations that request exclusion** — We classify the organizations that have requested exclusion from future scans.

Country	Organizations	
United States	129	(63.0%)
United Kingdom	15	(7.4%)
Germany	12	(5.9%)
Australia	9	(4.4%)
Canada	7	(3.4%)
Other	32	(15.0%)

Table 9: **Excluded addresses by country** — We geolocate the organizations that have requested exclusion and find that the majority are in the United States.

Country	Orgs	Hosts	% Addr Space
United States	96	1,029,632	0.07%
Korea	8	43,008	0.03%
Canada	7	25,344	0.04%
Austria	7	225,024	0.40%
Great Britain	5	1,536	0.001%
Romania	5	3,072	0.03%
France	5	133,120	0.17%
Portugal	5	80,640	1.1%
India	4	1,280	0.002%
Russia	4	8,192	0.01%

Table 10: **Inaccessible hosts by country** — We geolocate the routed blocks that are no longer accessible to scanning hosts.

country, a different pattern emerges, because the removal of a single AS can greatly impact the availability within the region. For example, while only one organization in Nigeria blacklisted our subnet, this single rule blocked more than 1% of the country's IP space. A similar situation appears in Portugal, Ireland, Luxembourg, Honduras, Argentina, and Lithuania.

5.3 Organizations Requesting Exclusion

Another indicator of scan detection can be found in the scan exclusion requests that we receive. Over the course of our HTTPS scanning, we have received 208 exclusion requests—resulting in the removal of 5.4 million addresses from our study—0.15% of the public IPv4 address space. Of the excluded hosts, 1.46 million (28%) had previously been seen hosting HTTPS. In comparison, only 1% of IPv4 hosts respond on port 443. We present the types of organizations that have requested exclusion in Table 8 and countries in Table 9. As with the organizations that dropped scan traffic, the majority of requests originated from the United States. We only received four requests from Asia and Africa: one each from Taiwan, India, South Africa, and Japan.

In our prior work [19], we suggest that researchers post a website that explains the purpose of their scanning and that they coordinate with their local network administrators. In order to understand whether this information was useful to network operators and to revise our recommendations, we tracked how network operators contacted us. We find that almost 60% of emails were sent directly to our research team via the site hosted on the scan IPs, 17% were sent to the WHOIS abuse contact, and 12% were sent to our institution's security office (e.g. security@umich.edu). We show a breakdown of contact points in Table 6.

Our informational page has been viewed by 6,600 unique users with an average of 357 visitors per month. More than 90% of visitors used common web browsers (Chrome, Firefox, Internet Explorer, Safari, or Opera). Viewers primarily geolocated to the United States, Germany, United Kingdom, Canada, and Japan. The ratio of page views to complaints (approximately 1:30) suggests that many organizations are cognizant of our scanning activity and do not object to it.

5.4 Blacklisting Scope

While we expected that a small number of organizations would block our scan hosts, it is not immediately clear what network segment organizations would block. We scanned from an additional, unrelated /24 in our institutional AS and found that 38,648 (39%) of the hosts that we could not reach on port 443 are also unavailable from the unrelated /24 in our AS. In other words, 39% of organizations that blocked our dirty subnet blocked the entire /16 in which our scan subnet is located or blocked our entire AS. In terms of estimated total inaccessible address space, 338,944 addresses (18.7% of the addresses inaccessible in our scan subnet) are possibly unavailable from the entire AS.

5.5 Temporal Analysis of Scan Detection

We initially hypothesized that our scanning would cause observant networks to immediately blacklist our network or contact our research team. If this were the case, we would expect that network exclusion requests would plateau after several scans. Instead, we find that organizations are slowly continuing to blacklist our scan subnet or request exclusion more than two years after we began regular scanning. In order to estimate when users detected scanning and blacklisted the scan subnet, we analyzed our historical data on the HTTPS ecosystem and recorded the last time any IP address in each routed block responded.

As shown in Figure 10, there is no plateau in the number of blacklisted hosts or in the number of organizations that have requested removal. Instead, we find that organizations continue to freshly notice the scanning behavior and to blacklist us or request exclusion. Further, more than half of the organizations began starting dropping traffic after more than a year of daily scans. We suspect that the organizations that request exclusion or begin blocking traffic years later are not proactively noticing scan traffic, but rather happening upon log entries during other maintenance and troubleshooting.

5.6 Scan Detection Mechanisms

In order to understand how organizations detect scans, we categorized the emails requesting exclusion or alerting us of potential abuse. In 64 cases (31%), network operators included evidence that was copied directly from log files or otherwise explained how they detected our scanning.

In 50% of cases, network operators noticed scans in their firewall or IDS logs. However, in 22% of reports, operators did not detect scanning in a firewall, but rather in their web logs (primarily Apache or nginx), and in 16% of cases, administrators noticed our scanning as our HTTPS handshake appeared to be a malformed handshake in SSH or OpenVPN logs. We show a breakdown of detection mechanisms in Table 11.

5.7 Revised Recommendations

We further emphasize the importance of researchers serving an informational webpage given the high percentage of users who used this to find contact information and the high number of views by network operators. We also recommend that researchers notify the owners of

Detection Mechanism	Organizations	
Firewall logs	22	(34%)
Web server logs	14	(22%)
Intrusion detection system (IDS) logs	10	(16%)
Invalid SSH or OpenVPN handshake	10	(16%)
Public blacklists	2	(3%)
Other	6	(9%)

Table 11: **Scan detection methods** — We classify the type of evidence included in email requests to be excluded in order to understand how organizations detect scanning.

various other email accounts at the institution including postmaster and administrator, in addition to institutional help desks, departmental administrators, and IT officials.

We add the additional recommendation that researchers publish the subnet being used for their research. This allows organizations that decide to drop traffic a mechanism to blacklist the correct subnet instead of dropping traffic from the entire institution.

6 Future Work

While we shed light on the broad landscape of large horizontal scans, there remain several open questions surrounding scan detection and defensive mechanisms.

Correlating distributed scanners It remains an open research problem to detect and correlate distributed scanning events. While we are able to estimate broad patterns in scanning behavior, we excluded scanners that operate at under 10 pps or targeted fewer than 100 hosts in our darknet. This likely excludes slow, massively distributed scans [6, 15]. While there has been previous research on detecting distributed scanning, little work has applied these to darknet data, in order to understand the slow scans that are taking place. Similarly, our darknet is primarily composed of contiguous address space, which may be avoided by some operations. It reamains an open issue to analyze distributed network telescopes to determine whether attackers are avoiding large blocks of consistently unresponsive address space.

IPv6 scanning In this work, we focused on scanning within the IPv4 address space. Scanning the IPv6 address space efficiently remains an open problem, as does analyzing existing IPv6 scanning behavior.

Vertical scanning Our study focused on horizontal scanning — scanning a single port across a large number of hosts. We note that during this investigation, we also stumbled upon several cases of large vertical scanning operations, which deserve further attention.

Exclusion standards Blacklisting by external organizations indicates a lack of communication between researchers and network operators. This misalignment has lead to organizations dropping all traffic from institutional ASes, which may have other adverse impacts. There currently exists no standard for system operators to request exclusion. Further work is needed to develop a standard similar to HTTP's robots.txt to facilitate this communication.

Determining intent Given that the majority of scanning takes place from large hosting providers, it is oftentimes difficult to discern the intent of the scanner beyond scanned protocol. Follow-up work is necessary to determine the follow-up actions of these scanners. Given that these large scans are happening from a small number of hosts, it may be possible to determine owners and track from where these attacks are originating. Automated mechanisms for signaling benign intent (such as centrally maintained whitelists) could help network operators distinguish between harmful and beneficial instances of wide-scale scanning.

Understanding defensive reactions We find that a minuscule number of organizations are dropping scan traffic. However, it is unclear whether other organizations are aware of and deliberately permit this research-focused traffic, or whether they are entirely unaware of it. More investigation is needed to understand the attentiveness of these organizations.

7 Conclusion

In this work, we analyzed the current practice of Internet-wide scanning, finding that large horizontal scanning is common and is responsible for almost 80% of non-Conficker scan traffic. We analyzed who is scanning and what services they are targeting noting differences from previously reported results. Ultimately, we find that researchers and attackers are both taking advantage of new scanning tools and hosting options — adapting to new advances in technology in order to further reduce the burden for finding vulnerabilities. While the landscape of scanning is evolving, defenders have remained sluggish in detecting and responding to even the most obvious scans.

Acknowledgments

We thank Paul Royal, Adam Allred, and their team at the Georgia Institute of Technology, as well as Thorsten Holz, Christian Rossow, and Marc Kührer at Ruhr-Universität Bochum for facilitating scans from their institutions. We similarly thank the exceptional sysadmins at the University of Michigan for their help and support throughout this project. This research would not have been possible without Kevin Cheek, Chris Brenner, Laura Fink, Paul

Howell, Don Winsor, and others from ITS, CAEN, and DCO. The authors thank Michael Kallitsis and Manish Karir of Merit Network for helping facilitate our darknet analysis. We additionally thank David Adrian, Brad Campbell, Jakub Czyz, Jack Miner III, Pat Pannuto, Eric Wustrow, and Jing Zhang. This work was supported in part by the Department of Homeland Security Science and Technology Directorate under contracts D08PC75388, FA8750-12-2-0235, and FA8750-12-2-0314; the National Science Foundation under contracts CNS-0751116, CNS-08311174, CNS-091639, CNS-1111699, CNS-1255153, and CNS-1330142; and the Department of the Navy under contract N000.14-09-1-1042.

References

[1] Backdoor found in Linksys, Netgear routers. https://news.ycombinator.com/item?id=6997159.

[2] Hackers spend Christmas break launching large scale NTP-reflection attacks. http://www.symantec.com/connect/blogs/hackers-spend-christmas-break-launching-large-scale-ntp-reflection-attacks.

[3] Open resolver project. http://openresolverproject.org/.

[4] Shadowserver open resolver scanning project. https://dnsscan.shadowserver.org/.

[5] M. Allman, V. Paxson, and J. Terrell. A brief history of scanning. In *Proc. 7th ACM SIGCOMM conference on Internet measurement*, pages 77–82, 2007.

[6] Anonymous. Internet census 2012. http://census2012.sourceforge.net/paper.html, Mar. 2013.

[7] L. Bello. DSA-1571-1 OpenSSL — Predictable random number generator, 2008. Debian Security Advisory. http://www.debian.org/security/2008/dsa-1571.

[8] A. J. Bonkoski, R. Bielawski, and J. A. Halderman. Illuminating the security issues surrounding lights-out server management. In *Proc. 7th USENIX Workshop on Offensive Technologies*. USENIX, 2013.

[9] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow. Elliptic curve cryptography in practice. In *Proc. 18th International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[10] S. Checkoway, M. Fredrikson, R. Niederhagen, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of dual EC in TLS implementations.

[11] R. Chirgwin. Hacker backdoors Linksys, Netgear, Cisco and other routers. http://www.theregister.co.uk/2014/01/06/hacker_backdoors_linksys_netgear_cisco_and_other_routers/.

[12] J. Conway. Ecatel's harboring of spambots and malware causes BGP peers to stop peering with them. http://www.sudosecure.com/ecatels-harboring-of-spambots-and-malware-causes-bgp-peers-to-stop-peering-with-them/.

[13] E. Cooke, M. Bailey, Z. M. Mao, D. Watson, F. Jahanian, and D. McPherson. Toward understanding distributed blackhole placement. In *Proc. ACM workshop on Rapid malcode*, pages 54–64, 2004.

[14] J. Czyz, K. Lady, S. G. Miller, M. Bailey, M. Kallitsis, and M. Karir. Understanding IPv6 Internet background radiation. In *Proc. 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.

[15] A. Dainotti, A. King, F. Papale, A. Pescape, et al. Analysis of a /0 stealth scan from a botnet. In *Proc. 12th ACM SIGCOMM Conference on Internet Measurement*.

[16] Z. Durumeric, D. Adrian, M. Bailey, and J. A. Halderman. Heartbleed bug health report. https://zmap.io/heartbleed/.

[17] Z. Durumeric and J. A. Halderman. Internet-wide scan data repository. https://scans.io.

[18] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *Proc. 13th ACM SIGCOMM Internet Measurement Conference*, Oct. 2013.

[19] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proc. 22nd USENIX Security Symposium*, Aug. 2013.

[20] P. Eckersley and J. Burns. An observatory for the SSLiverse. Talk at Defcon 18 (2010). https://www.eff.org/files/DefconSSLiverse.pdf.

[21] S. Gallagher. Backdoor in wireless DSL routers lets attacker reset router, get admin. http://arstechnica.com/security/2014/01/backdoor-in-wireless-dsl-routers-lets-attacker-reset-router-get-admin/.

[22] C. Gates. Coordinated scan detection, 2009.

[23] R. Graham. MASSCAN: Mass IP port scanner. https://github.com/robertdavidgraham/masscan.

[24] J. Green, D. J. Marchette, S. Northcutt, and B. Ralph. Analysis techniques for detecting coordinated attacks and probes. In *Workshop on Intrusion Detection and Network Monitoring*, pages 1–9, 1999.

[25] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proc. 21st USENIX Security Symposium*, Aug. 2012.

[26] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: A thorough analysis of the X.509 PKI using active and passive measurements. In *11th ACM SIGCOMM conference on Internet measurement (IMC)*, 2011.

[27] V. Jacobson, C. Leres, and S. McCanne. libpcap. Lawrence Berkeley National Laboratory, Berkeley, CA. Initial release June 1994.

[28] M. Javed and V. Paxson. Detecting stealthy, distributed SSH brute-forcing. In *Proc. ACM SIGSAC conference on Computer & communications security*, pages 85–96, 2013.

[29] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Symposium on Security and Privacy*, pages 211–225, 2004.

[30] J. Kasten, E. Wustrow, and J. A. Halderman. CAge: Taming certificate authorities by inferring restricted scopes. In *17th International Conference on Financial Cryptography and Data Security (FC)*, 2013.

[31] C. Leckie and R. Kotagiri. A probabilistic approach to detecting network scans. In *Network Operations and Management Symposium (NOMS)*, pages 359–372, 2002.

[32] D. Leonard and D. Loguinov. Demystifying service discovery: Implementing an Internet-wide scanner. In *Proc. 10th ACM SIGCOMM conference on Internet measurement (IMC)*, pages 109–122, 2010.

[33] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[34] J. C. Matherly. SHODAN the computer search engine, Jan 2009. http://shodanhq.com.

[35] MaxMind, LLC. GeoIP, 2013. http://www.maxmind.com/en/city.

[36] N. Mehta and Codenomicon. The heartbleed bug. http://heartbleed.com/.

[37] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. *Network telescopes: Technical report*. Department of Computer Science and Engineering, University of California, San Diego, 2004.

[38] H. Moore. Security flaws in universal plug and play. Unplug. Don't Play, Jan. 2013. http://community.rapid7.com/servlet/JiveServlet/download/2150-1-16596/SecurityFlawsUPnP.pdf.

[39] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet background radiation. In *Proc. 4th ACM SIGCOMM Conference on Internet Measurement*, pages 27–40, 2004.

[40] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. *SRI International*, 2009.

[41] M. Prince. Technical details behind a 400Gbps NTP amplification DDoS attack. http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack.

[42] C. Rossow. Amplification hell: Revisiting network protocols for DDoS abuse. In *Proc. Network and Distributed System Security Symposium*, Feb. 2014.

[43] S. E. Schechter, J. Jung, and A. W. Berger. Fast detection of scanning worm infections. In *Recent Advances in Intrusion Detection*, pages 59–81. Springer, 2004.

[44] E. Vanderbeken. TCP-32764: some codes and notes about the backdoor listening on tcp-32764 in linksys WAG200G. https://github.com/elvanderb/TCP-32764.

[45] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston. Internet background radiation revisited. In *Proc. 10th ACM SIGCOMM Conference on Internet Measurement*, pages 62–74. ACM, 2010.

[46] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of internet sinks for network abuse monitoring. In *Recent Advances in Intrusion Detection*, pages 146–165. Springer, 2004.

[47] J. Zhang, Z. Durumeric, M. Bailey, M. Karir, , and M. Liu. On the mismanagement and maliciousness of networks. In *Proc. Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.

On the Feasibility of Large-Scale Infections of iOS Devices

Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon Chung, Billy Lau, and Wenke Lee

School of Computer Science, College of Computing, Georgia Institute of Technology

{*tielei.wang, yeongjin.jang, yizheng.chen, pchung, billy, wenke*}*@cc.gatech.edu*

Abstract

While Apple iOS has gained increasing attention from attackers due to its rising popularity, very few large scale infections of iOS devices have been discovered because of iOS' advanced security architecture. In this paper, we show that infecting a large number of iOS devices through botnets is feasible. By exploiting design flaws and weaknesses in the iTunes syncing process, the device provisioning process, and in file storage, we demonstrate that a compromised computer can be instructed to install Apple-signed malicious apps on a connected iOS device, replace existing apps with attacker-signed malicious apps, and steal private data (e.g., Facebook and Gmail app cookies) from an iOS device. By analyzing DNS queries generated from more than half a million anonymized IP addresses in known botnets, we measure that on average, 23% of bot IP addresses demonstrate iOS device existence and Windows iTunes purchases, implying that 23% of bots will eventually have connections with iOS devices, thus making a large scale infection feasible.

1 Introduction

As one of the most popular mobile platforms, Apple iOS has been successful in preventing the distribution of malicious apps [23, 32]. Although botnets on Android and jailbroken iOS devices have been discovered in the wild [40, 42, 43, 45, 54], large-scale infections of non-jailbroken iOS devices are considered extremely difficult for many reasons.

First, Apple has powerful revocation capabilities, including removing any app from the App Store, remotely disabling apps installed on iOS devices, and revoking any developer certificate. This makes the removal of malicious apps relatively straightforward once Apple notices them.

Second, the mandatory code signing mechanism in iOS ensures that only apps signed by Apple or certified by Apple can be installed and run on iOS devices. This significantly reduces the number of distribution channels of iOS apps, forcing attackers to have their apps signed by a trusted authority.

Third, the Digital Rights Management (DRM) technology in iOS prevents users from sharing apps among arbitrary iOS devices, which has a side effect of limiting the distribution of malicious apps published on the App Store. Although recent studies show that malicious apps can easily bypass Apple's app vetting process and appear in the Apple App Store [26, 36, 51], lacking the ability to self-propagate, these malicious apps can only affect a limited number of iOS users who *accidentally* (e.g., by chance or when tricked by social-engineering tactics) download and run them. Specifically, to run an app signed by Apple, an iOS device has to be authenticated by the Apple ID that purchased the app. For example, suppose we use an Apple ID_A to download a copy of a malicious app from the App Store and later we install this copy on an iOS device that is bound to Apple ID_B. This copy cannot run on the iOS device that is bound to Apple ID_B because of the failure of DRM validation. On iOS 6.0 or later, when launching this app, iOS will pop up a window (Figure 1) to ask the user to re-input an Apple ID and a password. If the user cannot input the correct Apple ID (i.e., Apple ID_A) and the corresponding password, iOS refuses to run the app.

In this paper, we show that despite these advanced security techniques employed by iOS, **infecting a large number of non-jailbroken iOS devices through botnets is feasible**. Even though iOS devices are designed for mobile use, they often need to be connected to personal computers via USB or Wi-Fi for many reasons, such as backup, restore, syncing, upgrade, and charging. We find that the USB and Wi-Fi communication channels between iOS devices and computers are not well protected. Consequently, a compromised computer (i.e., a bot) can be easily instructed to install malicious apps onto its connected iOS devices and gain access to users'

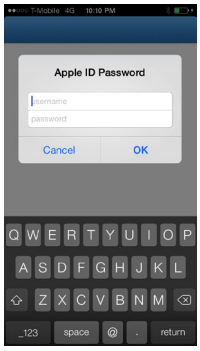


Figure 1: User attempting to run an app downloaded by a different Apple ID on his iOS device needs to first enter the correct Apple ID and password.

private data without their knowledge. In this paper, connected iOS devices refer to those that are plugged into a computer through USB cable or have Wi-Fi syncing enabled. Note that if Wi-Fi syncing is enabled, the iOS device will automatically sync with a paired computer when they are on the same network.

The feasibility of a large scale infection is facilitated by two main problems that we have discovered in our research. The first is a previously unknown design flaw in the iTunes syncing mechanism (including both USB and Wi-Fi based syncing), which makes the iTunes syncing process vulnerable to Man-in-the-Middle (MitM) attacks. By exploiting this flaw, attackers can first download an Apple-signed malicious app (e.g., a Jekyll app [51]) using their Apple ID and then remotely instruct a compromised computer to install the attacker's copy on a connected iOS device, completely bypassing DRM checks. In other words, an attacker can have a malicious app of his own choosing run on a user's iOS device without the user ever seeing an authentication pop-up window.

Coupled with botnet infrastructures, this exploit enables large scale delivery of arbitrary Apple-signed apps. This has two serious security implications. First, it challenges the common belief that the Apple App Store is the sole distributor of iOS apps. Instead of relying on tricking a user into downloading apps from the App Store, attackers can now push copies of their app onto a victim's device. Even if an app has been removed from the App Store, attackers can still deliver it to iOS users. Second, this expolit challenges the common belief that the installation of iOS apps must be approved by the user. Attackers can surreptitiously install any app they downloaded onto victim's device.

The second security issue we discovered is that an iOS device can be stealthily provisioned for development through USB connections. This weakness allows a compromised computer to arbitrarily remove installed third-party apps from connected iOS devices and install any app signed by attackers in possession of enterprise or individual developer licenses issued by Apple. This weakness leads to many serious security threats. For example, attackers can first remove certain targeted apps (such as banking apps) from iOS devices and replace them with malicious apps that look and feel the same. As a result, when a victim tries to run a targeted app, they actually launch the malicious app, which can trick the user to re-input usernames and passwords. We originally presented this attack [31] in 2013 and Apple released a patch in iOS 7 that warns the user when connecting iOS devices to a computer *for the first time*. However, this patch does not protect iOS devices from being stealthily provisioned by a compromised computer that the user already trusts.

In addition to injecting apps into iOS devices, attackers can also leverage compromised computers to obtain credentials of iOS users. Specifically, since many iOS developers presume that the iOS sandbox can effectively prevent other apps from accessing files in their apps' home directories, they store credentials in plaintext under their apps' home directories. For example, the Starbucks app was reported to save usernames and passwords in plaintext. Starbucks thought the possibility of a security exploit to steal the plaintext passwords was "very far fetched" [8]. However, once an iOS device is connected to a computer, all these files are accessible by the host computer. Consequently, malware on the computer can easily steal the plaintext credentials through a USB connection. In our work, we found that the Facebook and Gmail apps store users' cookies in plaintext. By stealing and reusing the cookies from connected iOS devices, attackers can gain access to the victims' accounts remotely.

While it is known that a host computer can partially access the file system of a connected iOS device, we point out that it leads to security problems, especially when attackers control a large number of personal computers. Considering that there are many apps dedicated to iOS, this problem may allow attackers to gain credentials that are not always available on PCs.

To quantitatively show that botnets pose a realistic threat to iOS devices, we also conduct a large scale measurement study to estimate how many compromised computers (i.e., bots) could connect with iOS devices. Intuitively, given the immense popularity of iOS devices and compromised Windows machines, we presume that many people are using iOS devices connected to compromised computers. However, to the best of our knowledge, there exists no previous work that can provide large scale measurement results.

By analyzing DNS queries generated from 473,506

Infection Type	Root Cause	Connection Type
Install malicious apps signed by Apple	Man-in-the-Middle attacks against syncing	USB or Wi-Fi
Install malicious apps signed by attackers	Stealthy provisioning of devices	USB
Steal private data (e.g., Facebook and Gmail apps's cookies)	Insecure storage of cookies	USB

Table 1: Infection Summary.

anonymized IP addresses[1] that were involved in known botnets on 10/12/2013 in 13 cities of two large ISPs in the US, we identified 112,233 IP addresses that had App Store purchase traffic issued by iTunes on Windows, as well as network traffic generated by iOS devices. This implies that the iOS users in the 112,233 home networks were purchasing items in the App Store from compromised Windows PCs on the same day. We make the following assumption: if iTunes is installed on a user's personal computer and is also used to purchase some item from the App Store, the user will eventually connect his or her iOS device(s) to it either via USB or Wi-Fi. Based on this assumption, we estimate that iOS devices in the 112,233 IP addresses could be infected via a connected computer. In other words, 23% of bots in our measured botnet could be used to infect iOS devices.

A broader implication of our study is that it may raise concerns about the security of mobile two-factor-authentication schemes [13]. In such schemes, a mobile phone is used as a second factor of authentication for transactions initiated on a potentially compromised PC. A fundamental assumption made by such schemes is that the "two factors" (i.e., the PC and the phone) are very hard to be reliably and simultaneously compromised (and linked to the same user) by an adversary. This assumption is reasonable if the PC and the phone are exposed to independent attack vectors. However, as shown in this paper, since a large number of users would connect mobile phones to their PCs, the PC itself becomes an attack vector to the phone. As such, the aforementioned assumption becomes dubious. An attacker who already controls the PC can now use it as a stepping stone to inject malware into the phone, and thus can control both factors. As a result, the attacker can easily launch attacks described in [21, 39] to defeat mobile two-factor-authentication schemes[2].

In summary, the main contributions of our work are:

- We discover a design flaw in the iTunes syncing process, and present a Man-in-the-Middle attack that enables attackers to run any app downloaded by their Apple ID on iOS devices that are bound to different Apple IDs, bypassing DRM protections. Based on the MitM attack, we present a way to deliver Apple-signed malicious apps such as Jekyll apps to iOS users.

- We point out the security implications of the stealthy provisioning process and insecure credential storage, and demonstrate realistic attacks, such as replacing installed apps in the iOS device with malicious apps and stealing authentication cookies of the Facebook and Gmail apps.

- We show that a large scale infection of iOS devices is a realistic threat and we are the first to show quantitative measurement results. By measuring iTunes purchases and iOS network traffic generated from IP addresses involved in known botnets, we estimate that on average, 23% of all bot population have connections with iOS devices.

Table 1 summaries the attacks. We have made a full disclosure to Apple and notified Facebook and Google about the insecure storage of cookies in their apps. Apple acknowledged that, based on our report, they have identified several areas of iOS and iTunes that can benefit from security hardening.

The rest of the paper is organized as follows. In Section 2, we demonstrate installation of Apple-signed malicious apps without violating DRM checks. In Section 3, we demonstrate replacing targeted apps with attacker-signed malicious apps. In Section 4, we demonstrate theft of private data (e.g., Facebook and Gmail apps' cookies) from iOS devices. In Section 5, we describe our measurement techniques that indicate large scale infection is feasible. Finally, we provide related work, discussion, and a conclusion.

2 Delivery of Apple-Signed Malicious Apps

This section discusses how a compromised computer can be instructed to install Apple-signed malicious apps on iOS devices. We explore the iOS DRM technology and iTunes syncing process in Section 2.1, present a Man-in-the-Middle attack in Section 2.2, and discuss how to bypass iOS DRM validations in Section 2.3.

[1]The original IP addresses were hashed into anonymized client IDs in our dataset. We performed our measurement over five days of DNS query data, and we used statistics from one day as an example here. We use IP address and client ID interchangeably.

[2]The ultimate defeat of mobile two-factor-authentication schemes will depend on what capabilities the injected mobile malware has. We further discuss it in Section 6.

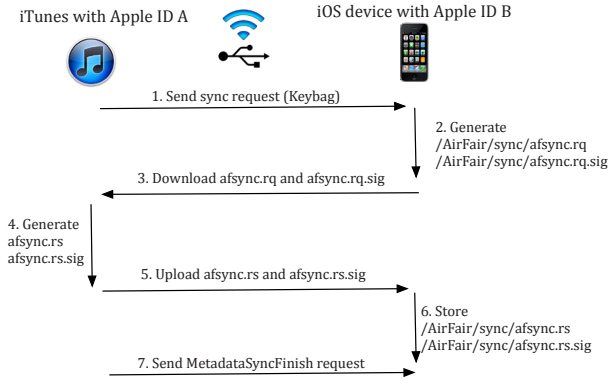


Figure 2: iTunes can sync apps to an iOS device with a different Apple ID.

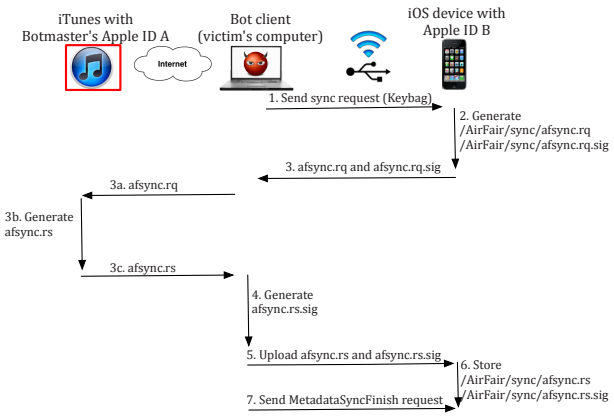


Figure 3: The Man-in-the-Middle against the syncing.

2.1 Fairplay DRM and iTunes Syncing

Apple utilizes a DRM (Digital Rights Management) technology called Fairplay to prevent piracy of iOS apps. All apps in the App Store are encrypted and signed by Apple. To run an app downloaded from the App Store, iOS will 1) verify an app's code signature, 2) perform DRM validation and decrypt the executable file, and 3) run the decrypted code. As a result, a copy of an app purchased by Apple ID_A cannot run on other iOS devices bound to other Apple IDs, because of the failure of DRM validation in step 2.

Although Apple does not disclose any technical details about the Fairplay DRM technology, we found a way to bypass it based on the following key observations.

- **Observation 1: Different Apple IDs will receive the same encrypted executable files for different copies of the same app**. After purchasing an app, an iOS user will receive a file with the `.ipa` extension from the App Store. The `ipa` file is in compressed zip format. We can retrieve the contents of an app package by decompressing the `ipa` file. The following shows the typical structure of an app directory (taking the Twitter app as an example).

```
/iTunesArtwork
/iTunesMetadata.plist
/Payload/Twitter.app/Twitter
...
/Payload/Twitter.app/SC_Info/Twitter.sinf
/Payload/Twitter.app/SC_Info/Twitter.supp
...
```

  Although the whole `ipa` package is unique for each Apple ID, we noticed that the encrypted executable files inside these `ipa` files are the same. Different copies of the same app purchased by different Apple IDs sharing same encrypted executable files implies that **the final decryption of the executables is irrelevant to Apple IDs**[3].

- **Observation 2: Apps purchased by different Apple IDs can run on the same iOS device under certain circumstances**. We found that iTunes can sync apps in its app library to iOS devices through USB or Wi-Fi connections, even if the iOS devices are bound to different Apple IDs. Specifically, when an iOS device with Apple ID_B is connected to iTunes with Apple ID_A, iTunes can still sync apps purchased by Apple ID_A to the iOS device, and authorize the device to run the apps. As a result, apps purchased by both Apple ID_A and Apple ID_B can run on the iOS device.

In particular, we reverse engineered the iTunes authorization process, i.e., how iTunes authorizes an iOS device with a different Apple ID to run its apps. We briefly present the workflow here.

First, iTunes sends a `Keybag` sync request to the iOS device (Step 1 in Figure 2). We refer the readers to [29] for detailed use of "Keybags" in iOS. For example, a keybag named Escrow is used for iTunes syncing and allows iTunes to back up and sync without requiring the user to enter a passcode.

Next, the iOS device generates an authorization request file `/AirFair/sync/afsync.rq` and corresponding signature file `/AirFair/sync/afsync.rq.sig` (Step 2 in Figure 2). Upon retrieving these two files from the iOS device (Step 3), iTunes generates an authorization response file `afsync.rs` and corresponding signature file `afsync.rs.sig` (Step 4).

iTunes then uploads the authorization response and signature files (`afsync.rs` and `afsync.rs.sig`) to the iOS device (Step 5). The iOS device stores the two files

[3] We further explain this in Section A.1

in the directory `/AirFair/sync/` and updates its internal state (Step 6).

Finally, iTunes sends a request to the iOS device to finish the syncing process (Step 7). After that, all apps in the iTunes app library can directly run on the iOS device without showing the pop-up window similar to Figure 1, even though iTunes and the iOS device are bound to different Apple IDs.

2.2 Remote Authorization

By reverse engineering the iTunes executables, we identified the functions in iTunes that are used to generate the `afsync.rs` and `afsync.rs.sig` (i.e., Step 4 in Figure 2). Based on these findings, we realized that by launching a Man-in-the-Middle-style attack, iTunes can remotely authorize an iOS device, without requiring physical connections between iTunes and the iOS device.

Figure 3 shows the remote authorization process in which iTunes is running on a remote computer (iTunes with Botmaster's Apple ID) and an iOS device is connected to a local computer (i.e., the bot client) through a USB cable or Wi-Fi connection. The local computer acts as a middleman.

First, the local computer, as instructed by the botmaster, sends a `Keybag` syncing request to a connected iOS device (Step 1 in Figure 3). After receiving the request, the iOS device generates the authorization request and signature file (Step 2), and transfers them to the local computer (Step 3). However, unlike the traditional iTunes authorization process, the local computer does not directly produce the authorization response file. Instead, it sends the authorization request file `afsync.rq` to a remote computer where iTunes is running (Step 3a). Upon receiving `afsync.rq`, the remote computer can force its local iTunes to handle the authorization request file as if it were from a connected iOS device, generate the authorization response file `afsync.rs` (Step 3b), and then send `afsync.rs` back (Step 3c).

Next, the local computer further produces the signature file `afsync.rs.sig` by using local iTunes code (Step 4). Note that the signature file is a keyed hash value of the response file using the connection session ID as the key. The signature file could also be generated by the remote iTunes if the local computer transfers the connection session ID to the remote iTunes in Step 3a. Furthermore, the local computer uploads `afsync.rs` received from the remote iTunes and `afsync.rs.sig` to the connected iOS device (Step 5). Step 6 and Step 7 in Figure 3 are the same as those in Figure 2.

The end result is that the iOS device connected to a local computer obtains authorization to run apps purchased by the iTunes instance running on a remote computer.

2.3 Delivery of Jekyll Apps

Background. Our previous work [51] demonstrated that malicious third-party developers can easily publish malicious apps on the App Store. It also implemented a proof-of-concept malicious app named Jekyll that can carry out a number of malicious tasks on iOS devices, such as posting tweets and dialing any number. Other research also demonstrated malicious keylogger apps on iOS 7.0.6 [55]. However, a key limitation of these apps is that attackers have to passively wait for iOS users to download the apps and thus affect only a limited number of iOS users.

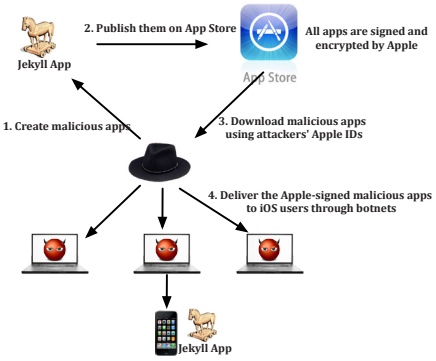


Figure 4: Deliver Jekyll apps to iOS devices.

Delivery Process. Figure 4 illustrates the high level workflow of the attack. First, attackers create Jekyll-like malicious apps using methods proposed in [51] and publish them on the App Store (Step 1 & 2 in Figure 4). Next, attackers download Jekyll-like apps from the App Store using Apple IDs under their control (Step 3). Finally, attackers deliver the downloaded apps to compromised computers and instruct them to install the downloaded apps to connected iOS devices.

Since Jekyll [51] has been removed from the App Store, we reused the copy of the app downloaded by our testing accounts. By using the remote authorization technique we described in Section 2.2, we successfully made our copy of Jekyll run on iOS devices bound to different Apple IDs without triggering DRM violations. The attack demonstrates that even if an app has been removed from the App Store, attackers can still distribute their own copies to iOS users. Although Apple has absolute control of the App Store, attackers can leverage MitM to build a covert distribution channel of iOS apps.

3 Delivery of Attacker-Signed Apps

In this section, we present how the USB interface can be exploited to install arbitrary apps that are signed by attackers. The iOS mandatory code signing mechanism ensures that only apps signed by trusted authorities can

run on iOS devices. Apple allows developers to install apps into iOS devices through a process called device provisioning, which delegates code signing to iOS developers. This was originally intended for developers to either test their apps on devices or for enterprises to do in-house app distribution. Unfortunately, we found that this process can be abused to install malicious apps into iOS devices.

3.1 Provisioning Process

Preparation. A provisioning profile is a digital certificate that establishes a chain of trust. It describes a list of iOS devices that are tied to an Apple ID, using the Unique Device Identifier (UDID) of each device. After sending the UDID of an iOS device to Apple, a provisioning profile is produced for installation on the device. Once installed, apps created and signed by the Apple ID that created the provisioning profile can be successfully executed in a provisioned iOS device. Although a device's UDID is considered sensitive information, it is straightforward for compromised machines to obtain the UDID of a connected iOS device because an iOS device exposes its UDID through the USB device descriptor header field.

Installing Provisioning Profiles. Conventionally, the provisioning process is transparently done when one is using Xcode (Apple's IDE). However, we found that the installation of provisioning profiles can also be done by directly sending requests to a service running on iOS devices called `com.apple.misagent`, launched via the `lockdownd` service [33]. Specifically, by crafting requests of the following key value pairs encapsulated in plist format [1]: <`MessageType`, `Install`>, <`Profile`, the provisioning profile to be installed>, and <`ProfileType`, `Provisioning`>, we can stealthily install a provisioning profile in a USB-connected iOS device [33].

Installing and Removing Apps. The removal of an app is done by issuing an `Uninstall` command and app-id to a service on the device called `com.apple.mobile.installation_proxy`. The installation of an app is done by issuing an `Install` command and app-id to the same service, with the addition of the path to where the app package resides on the computer [33]. Note that using the `com.apple.mobile.installation_proxy` service to install or remove apps also works for non-provisioned iOS devices.

3.2 Attack Examples

As we will present in Section 5.7, we discovered that 4,593 (4%) of 112,233 potential victims queried mobile banking domains in a day (10/12/2013), implying that these devices are likely to have mobile banking apps installed. We implemented a proof-of-concept program that can check whether a plugged-in iOS device has banking apps installed and replaces them with malicious apps that look and feel the same, but trick the user to re-input usernames and passwords.

4 Stealing Credentials

In this section, we demonstrate that in addition to injecting apps into iOS devices, attackers can also leverage bots to steal credentials of iOS users.

4.1 Background

Cookie as Credential. Due to the common architecture of backend servers, the most frequently used credential types in the iOS app model are HTTP cookies [17] and OAuth Tokens [25, 27]. We focus on cookies, as they generally provide a full set of privileges for a particular app, whereas OAuth credentials are utilized to provide limited permissions.

Many iOS apps (such as Facebook and Gmail) use cookies as their authentication tokens. After first login to these apps, a cookie is generated by the server, transmitted to the device, and stored locally. Later, this cookie is presented to the server for accessing its contents via APIs (mostly `RESTful` HTTP).

The locations of the stored cookies are determined by the library that processes HTTP requests. The most commonly used library is called `NSURLConnection` [15], which is provided by the default iOS SDK. In addition, many third-party HTTP libraries [49, 50] are built on top of `NSURLConnection`. As a result, these libraries store the cookies in the same fixed locations as `NSURLConnection`. After analyzing the `NSURLConnection` library, we found that all cookies are stored inside an app's home folder by default.

Due to sandbox-based isolation, accessing files inside directories of other apps is prohibited in iOS. Since all cookies are stored separately per app, unless attackers can bypass the sandbox, a cookie can only be accessed by the corresponding app itself. Thus, storing cookies in each app directory is widely used by third-party developers and is considered to be hard to exploit [8].

4.2 Threat from the PC

This assumed difficulty of exploitation is incorrect when USB-based attacks are considered in the threat model. From a USB connection, a host computer can connect to an iOS device not only through the iTunes sync protocol, but also via the Apple File Connection (AFC) protocol [46]. AFC is designed to access media

files (e.g., images taken from the camera, recorded audio, or music files) through the USB cable. Furthermore, there exists a service maintained by `lockdownd` called `com.apple.mobile.house_arrest`, which provides access to iOS app directories through the AFC protocol. As a result, a computer has full USB-based access to the contents of `/private/var/mobile/Applications/*`.

Attacks. Credentials can be accessed via the `Cookie.binarycookies` file within an app's directory using AFC. Both the httpOnly cookie that is protected from JavaScript access and secure cookies that are only transferred through HTTPS connections can be recovered. Applying these cookies to the same apps in different devices allows attackers to log-in to the services of their victims.

Real Examples. As a proof of concept, we implemented a tool that can retrieve the cookies of Facebook and Gmail apps from a USB-connected iOS device, and transfer them to another computer.

By using these stolen cookies, we successfully logged in as the victim via the web services for both Facebook and Gmail. Although we only demonstrated the attacks against Facebook and Gmail, we believe that our finding affects a number of third-party apps that store cookies in the way similar to the Facebook and Gmail apps.

5 Measurement

In this section, we describe the methodology and datasets we use to determine a lower bound of the coexistence of iOS devices, App Store purchases made from Windows iTunes, and compromised Windows machines in home networks, with a goal to quantitatively show that a large number of users are likely to connect iOS devices to infected personal computers.

5.1 Overview

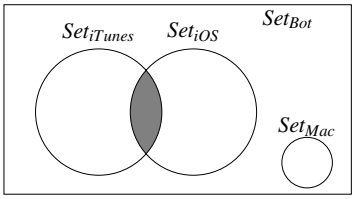


Figure 5: $(Set_{bot} - Set_{Mac}) \cap Set_{iOS} \cap Set_{iTunes}$ is the estimation of iOS devices that can be connected with bots.

Due to NAT, there are usually multiple machines (such as Windows machines, mobile phones, and Mac machines) behind a single client ID (i.e., an anonymized IP address). In our measurement, we used different fingerprints to determine the client IDs (CIDs) that produce network traffic generated by iOS devices, App Store purchases from Windows iTunes, and compromised Windows machines.

Identified as \ Truth	CID has Mac	CID has no Mac
CID has a Mac	True Positive	False Positive
CID has no Mac	False Negative	True Negative

Table 2: Definition of false positive and false negative.

First, we quantified the population of compromised Windows machines (i.e., bots) using a labeled C&C domains dataset (see Section 5.3). Since we only used C&C domains from Windows malware families, we are confident that all CIDs (Set_{bot} in Figure 5) have Windows machines. Next, within Set_{bot}, we further measured how many CIDs contain network traffic generated from iOS devices. Since there is an overlap of DNS queries from Mac OS X and queries from iOS, we excluded CIDs with Mac OS X traffic (Set_{Mac} in Figure 5). We used a set of iOS fingerprint domain names to identify a lower bound of CIDs containing iOS devices (Set_{iOS} in Figure 5). Then, we estimated how many iOS devices in Set_{iOS} are likely to be connected to a Windows machine. Since there is no unique DNS query generated when an iOS device is connected to a Windows machine either via USB or Wi-Fi, we assumed that if users have purchased an item from the App Store using Windows iTunes, they will eventaully connect iOS devices to a Windows machine. We estimated a lower bound of CIDs with Windows iTunes (Set_{iTunes} in Figure 5). Finally, the intersection of Set_{iOS} and Set_{iTunes} is our measurement result (shaded area in Figure 5).

We use false positives and false negatives to evaluate our methodology. They are defined in Table 2, using identification of Set_{Mac} as an example. If a CID has a Mac but we identify it as without a Mac, it is a false negative. On the other hand, if a CID does not have a Mac but we identify it as with a Mac, it is a false positive. The definition of false positive and false negative are similar to identifying Set_{iOS} and Set_{iTunes}. Since we want to be conservative about the size of $Set_{iTunes} \cap Set_{iOS}$, we accept false positives for Set_{Mac}, false negatives for Set_{iOS} and false negatives for Set_{iTunes}. However, we want to have small or zero false negatives for Set_{Mac} as well as false positives for Set_{iOS} and Set_{iTunes}.

5.2 Datasets

All datasets are provided by Damballa, Inc [3].

DNS Query Dataset. We obtained DNS traffic from two large ISPs in the US, collected in 13 cities for five days[4] in October 2013. A-type DNS queries from all clients to the Recursive DNS Servers of the ISPs were

[4] 10/12/2013, 10/24/2013, 10/27/2013, 10/28/2013, 10/30/2013.

captured by a sensor in each city, and then de-duplicated every hour to be stored. Each record in the dataset is a tuple of sensor ID, hourly timestamp, anonymized CID, query count, queried domain, and resolved IP address of the domain. The CID is a hash of original client IP address. The hash is performed to preserve client IP anonymity, and at the same time to retain a one-to-one mapping property. The query count denotes the number of times that client queried the particular domain and resolved IP pair in the hour. On average, we observed 54 million client IDs, 62 million queries, and 12 billion records daily from 13 sensors in total.

ISP Clients. Most CIDs in the two ISPs are home networks. Only few are small businesses. Since 99.5% of CIDs queried fewer than 1,000 distinct valid domains daily, we identified those as representative of home network users[5].

DHCP Churn. Because of long DHCP lease times (one week) and specific DHCP renewal policies (e.g., a modem's IP address will only be changed if some other modem has acquired the IP address after lease expiration) from the two large ISPs, the DHCP churn rate is very low in the networks we measured. Consequently, we consider a single CID as the same home in a given day. This is different than that of a botnet's perspective, where the bots could reside in any network [48].

Passive DNS Database. We collected and built a passive DNS database for the same days, from the same locations as the DNS query dataset to provide visibility of other types of DNS records. The database contains tuples of date, sensor ID, queried domain, query type, resolved data, and query count. Since there are only A-type records in the DNS query dataset, if there is a CNAME chain [38] for the domain name we want to measure, we use the passive DNS database to reconstruct the CNAME chain to determine the mapping of the domain we are interested in to the eventual returned A record. For example, the CNAME chain for Apple's iMessage server `static.ess.apple.com` is shown in Table 3. The final A-type domain for `static.ess.apple.com` is `e2013.g.akamaiedge.net`. If we want to know how many CIDs queried `static.ess.apple.com` in a day, we first examine the passive DNS database for that day to reconstruct all possible CNAME chains to `e2013.g.akamaiedge.net`. Next, we make sure that only `static.ess.apple.com` resolved into `static.ess.apple.com.edgekey.net`, and only `static.ess.apple.com.edgekey.net` resolved into `e2013.g.akamaiedge.net`. Finally, we measure `e2013.g.akamaiedge.net` in DNS query dataset.

HTTP Dataset. We utilized technology provided by Damballa to collect HTTP headers related to iOS and Windows iTunes, as well as those related to domain names of our selection. If User Agent strings related to iOS or Windows iTunes appeared in an HTTP header, or if any selected domain name was in the "Host" field of an HTTP request, we collected the request and corresponding response headers. The time frame for all HTTP headers we obtained is from 10/18/2013 to 11/11/2013. Since the HTTP data is not always available for all CIDs in our DNS query dataset, we only used the HTTP dataset to obtain ground truth for evaluating our approach.

Labeled C&C Domains We acquired all command and control (C&C) domain names for botnets that Damballa is tracking. The threat researchers in Damballa labeled those C&C domains using various methods, including static and dynamic analyses of malware, several public blacklists, historical DNS information, etc. We only picked malware families for Windows for the purpose of this measurement, since we wanted to avoid Mac OS X as much as possible.

`static.ess.apple.com. 3600 IN CNAME static.ess.apple.com.edgekey.net.`
`static.ess.apple.com.edgekey.net. 21600 IN CNAME e2013.g.akamaiedge.net.`
`e2013.g.akamaiedge.net. 20 IN A 23.73.152.93`

Table 3: CNAME chain for static.ess.apple.com.

5.3 Bot Population

First, we used labeled C&C domains to find CIDs containing infected Windows machines in the DNS query dataset. If a CID queried any C&C domain in a day, we consider it as having a bot at home for that day. During the five days in October, we observed 442,399 to 473,506 infected CIDs daily, with an average of 459,326. In particular, there were 473,506 infected CIDs on 10/12/2013. We use statistics from 10/12/2013 as an example below to explain how we determine the population of iOS devices and Windows iTunes.

5.4 Excluding Client IDs with Mac OS X

We utilized unique software update traffic to fingerprint Mac OS X. Apple lists five domain names related to OS X software update services [16]:

`swscan.apple.com`, `swquery.apple.com`, `swdist.apple.com`
`swdownload.apple.com`, `swcdn.apple.com`

Mac OS X is set to automatically check for security updates daily since version 10.6 [14] and to check for general software updates daily since version 10.8 [19]. We assume that if there is any Mac OS X within a CID, there must be a query to at least one of the five domains every day. According to [11], the percentage of OS X versions later than or including 10.6 is 95%. This assumption gives us a low false negative rate. To

[5] We explain why we use 1,000 as the threshold in Section A.2

evaluate the false positives of this approach, we verified using the HTTP dataset (See Table 5). We were able to collect HTTP headers for three of the five domain names. 3,530 headers for `swdist.apple.com`, 9,643 headers for `swscan.apple.com`, and 18,649 headers for `swcdn.apple.com` were observed. Among 115 unique (Source IP, User Agent string) pairs for `swdist.apple.com`, 114 User Agent strings were from Mac OS X, and one was from Mozilla, which gives us a $\frac{1}{115}$ false positive rate. Similarly, nine out of 3,884 pairs of (Source IP, User Agent string) for `swscan.apple.com` were false positives. We identified

Weather App	`apple-mobile.query.yahooapis.com`
Stocks App	`iphone-wu.apple.com`
Location Service	`gs-loc.apple.com` `iphone-wu.apple.com`

Table 4: Domains for finding iOS devices.

6,966 (1.50%) of 473,506 infected CIDs with Mac OS X on 10/12/2013. It is lower than the market share of Mac OS X since we only looked for CIDs infected with Windows malware. After excluding Mac OS X, we have 466,540 bot CIDs without Mac OS X, i.e., Set_{bot}-Set_{Mac}.

5.5 iOS Device Population

We used unique domains from two default apps and one service in iOS (the Weather app, Stocks app, and Location Services) to get a lower bound of CIDs containing iOS devices. We obtained these domains in Table 4 by capturing and analyzing network traffic when Weather, Stock, and Location Services were used in a controlled network environment. We also used the HTTP dataset for evaluation. As Table 5 shows, within all (Source IP, User Agent string) pairs that requested the three domains, all User Agent strings were from either iOS or Mac OS X. There were no User Agent strings from other operating systems. Since we have already excluded CIDs with Mac OS X traffic in the previous step, domains in Table 4 can be used to fingerprint iOS without introducing any false positives. However, if a user did not use Weather, Stocks, or Location Services in the day, it is a false negative.

Of 466,540 CIDs without Mac OS X traffic on 10/12/2013, 142,907 (Set_{iOS}) queried any of the three domains, indicating 30.63% of observed Windows bots have iOS devices in the same home network.

5.6 Windows iTunes Population

After we identified infected CIDs containing iOS devices, we further analyzed how many of these have iTunes installed in infected Windows. The biggest challenge here is that there is no unique domain name that can effectively fingerprint Windows iTunes, because Windows iTunes traffic is similar to the traffic generated by the App Store for iOS.

Fortunately, we found that because of the Apple Push Notification Service [2], iOS devices need to constantly query a certain domain name for push server configurations. Based on this feature, we define **iOS heartbeat DNS queries** as DNS queries that an iOS device always makes as long as it is connected to the Internet.

To pinpoint Windows iTunes, our observation is that if we observe App Store purchases but do not find iOS heartbeat DNS queries, then the purchases must originate from iTunes. Next, we describe how we identify the iOS heartbeat DNS queries and App Store purchase queries.

5.6.1 App Store Purchase

To fingerprint App Store purchases, we experimented with several App Store purchases in both Windows iTunes and iOS. By analyzing PCAP traces of these purchases, we discovered that domain names of the pattern `p*-buy.itunes.apple.com` is related to a purchase, where * denotes numbers. We used the HTTP dataset to check this pattern (See Table 5). 487 HTTP headers were collected from 10/26/2013 to 10/29/2013. From the 119 (Source IP, User Agent string) pairs, we confirmed that this pattern comes from the purchase of apps in either iOS or iTunes.

5.6.2 iOS Heartbeat DNS Queries

To discover iOS heartbeat DNS queries, we first collected all domains in the "Host" field of HTTP requests containing iOS-related and Windows iTunes-related User Agent strings from the HTTP dataset. Next, we examined domains that received a large number of requests, and concluded that the domain name `init-p01st.push.apple.com` is constantly queried for Apple push server configurations and certificates from iOS, but queried much less often by Windows iTunes.

Apple does not disclose how often iOS devices query their push server. To confirm our observed queries were iOS heartbeats, we utilized the following three methods.

1. HTTP Traffic Analyses: 8,990 HTTP headers were gathered for `init-p01st.push.apple.com`, from 10/18/2013 to 10/31/2013. By inspecting the distribution of "max-age" values of the Cache-Control field in the HTTP response headers, we were able to know the intended cache policy for the push server certificate in the response. For iOS, the observed max-age values were from 338s to 3,600s; for APSDaemon.exe (part of Windows iTunes), values ranged from 131,837s to 1,295,368s. Compared to Windows, iOS caches the push server certificate for a shorter time, with one hour maximum. Consequently, iOS devices must

Domain	Time Frame	HTTP headers	(Source IP, UA)	iOS	Mac	Mozilla	Other
swdist.apple.com	10/24-11/04/2013	3,530	115	0	114	1	0
swscan.apple.com	10/24-11/04/2013	9,643	3,884	0	3,875	9	0
swcdn.apple.com	10/19-11/11/2013	18,649	613	0	140	473	0
iphone-wu.apple.com	10/18-11/04/2013	17,606	1,772	1,174	598	0	0
apple-mobile.query.yahooapis.com	10/22-11/02/2013	16,808	3,018	2,999	19	0	0
gs-loc.apple.com	10/22-11/04/2013	2,380	561	367	182	0	0
p*-buy.itunes.apple.com	10/26-10/29/2013	487	119	-	-	-	App Store
init-p01st.push.apple.com	10/18-10/31/2013	8,990	-	-	-	-	-

Table 5: Ground truth of fingerprint domain names.

query `init-p01st.push.apple.com` when the cache expires.

2. Reverse Engineering: We also reverse engineered the push service daemon process `apsd` in iOS located at `/System/Library/PrivateFrameworks/ApplePushService.framework/`. We found that the URL `http://init-p01st.push.apple.com/bag` is used to set up the push-server's configuration path in the class `APSEnvironmentProduction` through the method `setConfigurationURL`. Furthermore, in another private framework called `PersistentConnection`, we found that the maximum length of time this connection can last is set to 1,800s by `setMaximumKeepAliveInterval(1800.0)`. This means iOS devices must re-send an HTTP request to get push server configurations at least every 1,800s. Moreover, the final A-type domain from `init-p01st.push.apple.com` has a TTL of 20s. As a result, every time the HTTP request is made, there must be a DNS query for the A-type domain.

3. Idle iPod Touch Experiment: We set up an iPod Touch to never auto-lock, connected it to a WiFi hotspot, and left it idle for 35.5 hours. During this period, there were 150 DNS queries to `init-p01st.push.apple.com` in total. Average query interval was 859s, with a maximum value of 1,398s and a minimum value of 293s. The maximum query interval is consistent with our 1,800s result via reverse engineering.

Our findings show that as long as an iOS device is connected to the Internet, it constantly queries `init-p01st.push.apple.com` for push service configurations, with query intervals shorter than an hour. Each query interval is determined by both the HTTP Cache-Control value and an enforced maximum interval value. The query interval for `init-p01st.push.apple.com` from iTunes is much longer. The reason for this query interval difference may be that iOS devices are more mobile than PCs. As an iOS device moves, it might need a push server closer to its current location, to ensure small push service delay.

Given the unique fingerprint for App Store purchases and a unique iOS heartbeat query pattern, we inferred a lower bound of Windows iTunes by estimating the number of CIDs that queried `p*-buy.itunes.apple.com` but did not query `init-p01st.push.apple.com` in the hour before and after the purchase query. If the Windows iTunes happened to query `init-p01st.push.apple.com` within those two hours, or if the user did not purchase anything in the iTunes App Store, it would be a false negative, since we cannot recognize Windows iTunes even though it exists. However, there are no false positives. Note that we cannot use `init-p01st.push.apple.com` to estimate the number of iOS devices for two reasons: i) Windows iTunes can query this domain; ii) the final A-type domain of `init-p01st.push.apple.com` can come from multiple original domain names[6]. Therefore, by removing CIDs that queried `init-p01st.push.apple.com` in the small time window, we exclude a larger set of CIDs that purchased from within iOS, resulting in a lower bound for Windows iTunes estimation.

On 10/12/2013, from the 142,907 infected CIDs with iOS devices, we further identified 112,233 CIDs with Windows iTunes purchases on the same day, i.e., $Set_{iTunes} \cap Set_{iOS}$. This indicates that 112,233 (23.70%) of CIDs have both iOS devices and Windows iTunes, but no Mac OS X within the home network.

5.7 Mobile Banking Traffic

The iOS devices behind bot CIDs with Windows iTunes are potential victims of our attacks. To estimate the percentage of those devices that may have banking apps installed, we chose mobile domains from eight banks (Citibank, Wells Fargo, PNC, Bank of America, Suntrust, Bank of the West, and U.S. Bank), and examined how many of those iOS devices queried them. We discovered that 4,593 (4%) of 112,233 potential victims queried mobile banking domains on 10/12/2013. This indicates that these devices are likely installed with mobile banking apps that could be replaced with fake mobile banking apps once they are infected, as discussed in Section 3.

[6]This is the only fingerprint domain name we used whose final A-type domain could be resolved from multiple different domain names.

5.8 Result Summary

We used the methodology described in this section to measure the number of iOS devices that can be potentially infected using the MitM attack described in Section 2, with five days of DNS query data. The results are summarized in Table 7. On average, we identified 459,326 bots daily. For 30% of bots, there exist iOS devices used from the same CID; and for 23% of all bots, there are both Windows iTunes installed and an iOS device used. Statistics for individual botnets as tracked by Damballa are presented in Table 6. For example, if the botmaster of botnet α bundled our attacks into their payload, there would be 75,714 potential victims in 13 cities, within the networks we monitor.

Botnet	Size	$Set_{bots} \cap Set_{iOS} \cap Set_{iTunes}$	Percentage
α	287,055	75,714	26.38%
β	69,895	12,517	17.91%
γ	49,138	10,216	20.79%
δ	16,236	3,232	19.91%
ε	13,732	2,662	19.39%
ε	5,024	1,182	23.53%
ζ	4,554	944	20.73%
η	4,377	929	21.22%
θ	4,231	834	19.71%
ϑ	4,067	806	19.82%

Table 6: Statistical analysis of the top 10 botnets with highest number of infected CIDs on 10/12/2013.

Date	Set_{bots}	$Set_{bots} \cap Set_{iOS}$	$Set_{bots} \cap Set_{iOS} \cap Set_{iTunes}$
10/12	473,506	142,907 (30.63%)	112,233 (23.70%)
10/24	452,003	134,838 (29.83%)	104,225 (23.06%)
10/27	442,399	134,271 (30.35%)	104,075 (23.53%)
10/28	461,144	138,793 (30.10%)	105,056 (22.78%)
10/30	467,579	141,242 (30.21%)	102,795 (21.98%)

Table 7: Measurement results summary, October 2013.

6 Related Work

USB Attack Vector. The USB interface has been demonstrated to be an attack vector for mobile devices for some years [18, 31, 35, 52]. Z. Wang and A. Stavrou [52] studied attacks that take advantage of USB interface connectivity and presented three attack examples on Android platforms that spread infections from phone to phone, from phone to computer, and from computer to phone. The work in [31, 35] further demonstrated that these USB based attacks can take place through USB charging stations or chargers. It is worth noting that infecting connected mobile devices from the PC has happened in the real world. Symantec has found malware samples on Windows that can inject malicious apps to USB-connected Android devices [6].

Our work is the first to show measurement results that indicate a large number of users are likely to connect iOS devices to compromised computers, potentially leading to a large scale infection of iOS devices. We hope that our measurement results could drive Apple and other mobile manufacturers to redesign the security model of USB connections, and remind app developers to securely store credentials. In addition, while most previous works focus on Android, we present various attacks against non-jailbroken iOS devices that can be launched via USB or Wi-Fi connections.

Attacks Against iOS. In recent years, more attacks against the iOS platform have been observed As one of the most representative attacks, jailbreaking is the process of obtaining root privilege and removing certain limitations (such as code signing) on iOS devices by exploiting vulnerabilities in the kernel, the boot loader, and even the hardware [37]. Since most jailbreaking tools [22, 30] deliver the exploits through a USB connection, attackers could also take advantage of these jailbreaking tools to root USB-connected iOS devices. In this case, attackers can easily inject malicious apps with the ability to read and send SMS (e.g., [5]), which will allow for more advanced attacks against SMS-based two factor authentication schemes [21, 39].

Many researchers have shown that the App Store cannot prevent publishing of malicious apps [26, 36, 51]. They also proposed defenses [20, 53] for jailbroken devices. As previously mentioned, these malicious apps could only affect a limited number of iOS users who downloaded them. Our research describes the means to deliver malicious apps to a significant number of iOS devices and could significantly amplify the threat of iOS-based malware.

Many works focus on reverse engineering iOS and its protocols. Researchers analyzed the iMessage protocol and proposed Man-in-the-Middle attacks [44]. Requiem [47] reverse engineered the Apple Fairplay DRM protection algorithm for music, movies, and eBooks, and can bypass Fairplay to decrypt protected media files. However, Requiem does not support Apple Fairplay DRM protection for apps in the App Store. The libimobiledevice project [33] enables a computer to communicate with USB-connected iOS devices without requiring iTunes, such as syncing music and video to the device and managing SpringBoard icons and installed apps. However, libimobiledevice does not contain the iTunes authorization process (Section 2.1). In other words, libimobiledevice can install an app purchased by Apple ID_A to an iOS device bound to Apple ID_B, but the app cannot successfully run due to the failure of DRM validations. In comparison, we analyzed the iTunes authorization process, and found a way to bypass the DRM validations. This can allow attackers to deliver malicious apps downloaded by their Apple IDs to different iOS devices.

Mobile OS Fingerprint. To fingerprint mobile OSes in a multi-device network environment, the User Agent field of the HTTP header, DHCP request header fields, Organization Unique Identifier (OUI, i.e., the first 3-bytes of a MAC address), or a combination of these were commonly used [12, 24, 28, 34, 41]. Unfortunately, it is not scalable to collect these data in ISP-level networks. Furthermore, it is common for each client IP in a cellular network to represent only one device. In ISP networks, many client IPs represent a NAT endpoint. To cope with the complexity caused by multiple devices per client IP in ISP networks, we used unique domains from two default apps and one service within iOS to measure the number of iOS devices. We also found a domain name related to the push service with a unique query frequency that allowed us to determine the presence or absence of an iOS device for a given client IP.

7 Discussion

7.1 Accuracy of the Measurement

We emphasize that the goal of our measurement study is to show that there is a large number of users who are likely to connect their iOS devices to compromised computers, which we argue may lead to a large scale infection of iOS devices through botnets. There are many reasons that may lead to underestimations in our measurement, which implies that even more iOS devices could be infected by the botnets that we monitor. For example, in our measurement, we did not consider cellular traffic and the case that people often have multiple iOS devices in their household.

Next, we discuss the potential reasons that may result in overestimations and analyze why they are unlikely to happen in our dataset.

Multiple Windows machines. The data we have only allows us to determine what *type* of devices are behind an IP address, but not how many of each. Thus, it is possible that there are multiple Windows machines behind an IP. In the case that there are multiple Windows machines and not all of them are infected, we may have an overestimated infection number since iOS devices could be connected to only the uninfected computers. To reduce this risk of overestimation, we excluded IPs that queried more than 1,000 unique domains in a day because these IPs are likely to be a gateway with many users. On the other hand, we can expect that within a small environment, if one Windows machine in a NAT environment is infected, it is likely that all Windows machines will eventually be infected. This is because 1) most likely all the Windows machines in the network are managed in a similar manner, and have the same level of updates and defenses, thus share the same vulnerabilities, and 2) it is likely that there is some kind of communication or data sharing (e.g., using the same Wi-Fi network or a USB thumb drive) between the machines.

Mobility of iOS devices. Due to the mobility of iOS devices, it is possible that the same iOS device appears in different "infected" IP addresses, which leads to an overestimation of the number of potential iOS victims. However, we argue that this is extremely unlikely in our dataset because the overestimation can only happen when the same iOS device is present behind different NAT-networks in the same ISP, and the NATs have infected computers that make purchases from Windows iTunes in the same day.

7.2 Mitigation and Prevention

Since Apple has remote removal and revocation abilities, they have complete mediation over what app can run on an iOS device. However, due to a significant number of apps on the App Store and the lack of runtime monitors on iOS devices, malicious apps are only detected when the users perceive adverse effects of the malicious apps. As a result, many iOS users may have already been affected by such attacks before the manually-triggered revocations and removal are applied. We have observed many Android botnets [42, 43] even though Google also has remote app removal ability [4]. In addition, this ability cannot prevent an attack that steals a user's cookies (as discussed in Section 4).

Since we only tested a few devices in the attack presented in Section 2, we cannot confirm whether Apple is able to impose a limit on the number of iOS devices that can be authorized per Apple ID. However, since registering an Apple ID only requires a valid email address, attackers can easily prepare a number of Apple IDs and use them to distribute malicious apps. Nonetheless, we still suggest that Apple should monitor the anomalous Apple IDs that deliver purchased apps to excessive number of devices. In addition, we advocate that iOS should warn the user when an app purchased by a different Apple ID is to be installed.

The attack in Section 3 relies on iOS developer licenses. An individual iOS developer license can only register up to 100 iOS devices, which prevents large scale exploitation. However, Apple has the enterprise developer license program, which allows provisioning of an arbitrary number of iOS devices. Although the application for an Enterprise iOS Developer License is arguably very difficult because one would require a Dun & Bradstreet (D-U-N-S) number, we have observed a lot of real world cases of abusing enterprise licenses, such as distributing pirated iOS apps [10], running GBA emulators [9], and delivering jailbreak exploits [7]. It is also

possible for attackers to obtain such licenses through infected machines under their control, rather than applying for one. As result, we suggest that iOS should warn the user when a provisioning profile is installed or prompts the user the first time an app is run that is signed by an unknown provisioning profile.

To prevent the malicious PC from stealing cookies through the USB connection, third-party developers should be aware that plaintext credentials could be easily leaked through the USB interface and store the credentials in a secure manner.

8 Conclusion

This paper discussed the feasibility of large scale infections of non-jailbroken iOS devices. We demonstrated three kinds of attacks against iOS devices that can be launched by a compromised computer: delivering Apple-signed malicious apps, delivering third-party developer signed malicious apps, and stealing private data from iOS devices. Furthermore, by analyzing DNS queries generated from more than half a million IP addresses in known botnets, we measured that on average, 23% of bots are likely to have USB connections to iOS devices, potentially leading to a large scale infection.

9 ACKNOWLEDGMENTS

We thank our anonymous reviewers and Manos Antonakakis for their invaluable feedback. We also thank the various members of our operations staff who provided proofreading of this paper. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] Apple property list. https://developer.apple.com/library/mac/documentation/cocoa/conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html.

[2] Apple push notification service. http://en.wikipedia.org/wiki/Apple_Push_Notification_Service.

[3] Damballa, inc. https://www.damballa.com/.

[4] Google throws kill switch on android phones. http://googlemobile.blogspot.com/2011/03/update-on-android-market-security.html.

[5] irealsms. http://irealsms.com/.

[6] New malware tries to infect android devices via usb cable. http://www.symantec.com/security_response/writeup.jsp?docid=2014-012109-2723-99.

[7] Pangu jailbreak for ios7.1.x. http://pangu.io/.

[8] Starbucks app leaves passwords vulnerable. http://money.cnn.com/2014/01/15/technology/security/starbucks-app-passwords/.

[9] Abusing enterprise distribution program to let users install gba emulator, 2013. http://www.iphonehacks.com/2013/07/apple-revokes-gba4ios-signing-certificate.html.

[10] Offering pirated ios apps without the need to jailbreak, 2013. http://www.extremetech.com/mobile/153849-chinese-app-store-offers -pirated-ios-apps-without-the-need-to-jailbreak.

[11] Operating system market share, 2014. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpsp=2014&qpnp=1&qptimeframe=Y.

[12] M. Afanasyev, T. Chen, G. M. Voelker, and A. C. Snoeren. Analysis of a mixed-use urban wifi network: when metropolitan becomes neapolitan. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, IMC '08, pages 85–98, New York, NY, USA, 2008. ACM.

[13] F. Aloul, S. Zahidi, and W. El-Hajj. Two factor authentication using mobile phones. In *IEEE/ACS Computer Systems and Applications*. 2009.

[14] Apple Inc. Mac OS X Snow Leopard and malware detection. http://support.apple.com/kb/HT4651, June 2011.

[15] Apple Inc. NSURLConnection Class Reference, Nov 2013. https://developer.apple.com/library/ios/documentation/cocoa/reference/foundation/Classes/NSURLConnection_Class/Reference/Reference.html.

[16] Apple Inc. Requirements for Software Update Service. http://support.apple.com/kb/ht3923, October 2013.

[17] A. Barth. HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265, April 2011.

[18] A. Bates, R. Leonard, H. Pruse, D. Lowd, and K. Butler. Leveraging usb to establish host identity using commodity devices. In *Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[19] K. Bell. Apple Increases Mountain Lion Security With Daily Update Checks, Automatic Installs, More, June 2012. http://www.cultofmac.com/175709/apple-increases-mountain-lion-security-with-daily-update-checks-automatic-installs-more/.

[20] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A. reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[21] A. Dmitrienko, C. Liebchen, C. Rossow, and A.-R. Sadeghi. On the (in)security of mobile two-factor authentication. Technical Report TUD-CS-2014-0029, CASED/TU Darmstadt, Mar. 2014.

[22] Evad3rs. Evasi0n jailbreaking tool. 2013. http://evasi0n.com/.

[23] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, pages 3–14, 2011.

[24] A. Gember, A. Anand, and A. Akella. A comparative study of handheld and non-handheld traffic in campus wi-fi networks. In *Proceedings of the 12th international conference on Passive and active measurement*, PAM'11, pages 173–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] E. Hammer-Lahav. The OAuth 1.0 Protocol. `http://tools.ietf.org/html/rfc5849`, April 2010.

[26] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. H. Deng, D. Gao, Y. Li, and J. Zhou. Launching generic attacks on ios with approved third-party applications. In *11th International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Banff, Alberta, Canada, June 2013.

[27] D. Hardt. The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`, October 2012.

[28] L. Invernizzi, S. Miskovic, R. Torres, S. Saha, S.-J. Lee, C. Kruegel, and G. Vigna. Nazca: Detecting Malware Distribution in Large-Scale Networks. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '14)*, Feb 2014.

[29] iOS Security, May 2012. `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`.

[30] Y. Jang, T. Wang, B. Lee, and B. Lau. Exploiting unpatched ios vulnerabilities for fun and profit. In *Black Hat USA*, 2014.

[31] B. Lau, Y. Jang, C. Song, T. Wang, P. H. Chung, and P. Royal. Mactans: Injecting malware into ios devices via malicious chargers. In *Black Hat USA*, 2013.

[32] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Proceedings of The 20th Annual Network and Distributed System Security Symposium*, 2013.

[33] Libimobiledevice. 2013. `http://www.libimobiledevice.org/`.

[34] G. Maier, F. Schneider, and A. Feldmann. A first look at mobile hand-held device traffic. In *Proceedings of the 11th international conference on Passive and active measurement*, PAM'10, pages 161–170, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] B. Markus, J. Mlodzianowski, and R. Rowley. Juice jacking, 2011. `http://www.techhive.com/article/238499/charging_stations_may_be_juice_jacking_data_from_your_cellphone.html`.

[36] C. Miller. Inside ios code signing. In *Symposium on Security for Asia Network (SyScan)*, Taipei, Nov 2011.

[37] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann. *iOS Hacker's Handbook*. Wiley, 1 edition, May 2012.

[38] P. Mockapetris. Domain Names - Concepts and Facilities. `http://www.ietf.org/rfc/rfc1034.txt`, November 1987.

[39] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert. Sms-based one-time passwords: Attacks and defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2013.

[40] C. Mulliner and J.-P. Seifert. Rise of the iBots: 0wning a telco network. In *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software (Malware)*, Nancy, France, October 2010.

[41] I. Papapanagiotou, E. M. Nahum, and V. Pappas. Smartphones vs. laptops: comparing web browsing behavior and the implications for caching. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 423–424, New York, NY, USA, 2012. ACM.

[42] V. Pidathala, H. Dharmdasani, J. Zhai, and Z. Bu. Misosms: one of the largest advanced mobile botnets. `http://www.fireeye.com/blog/technical/botnet-activities-research/2013/12/misosms.html`.

[43] H. Pieterse and M. Olivier. Android botnets on the rise: Trends and characteristics. In *Information Security for South Africa (ISSA)*, 2012.

[44] pod2g and gg. imessage privacy. In *Hack In The Box Amsterdam*, 2013.

[45] P. Porras, H. Sadi, and V. Yegneswaran. An analysis of the ikee.b iphone botnet. In *The Second International ICST Conference on Security and Privacy in Mobile Information and Communication Systems*, 2010.

[46] M. RENARD, 2013. Hacking apple accessories to pown iDevices.

[47] Requiem. 2013. `http://en.wikipedia.org/wiki/FairPlay#Requiem`.

[48] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[49] The Restkit Project. RestKit, Nov 2013. `http://restkit.org/`.

[50] M. Thompson. Afnetworking, Nov 2013. `https://github.com/AFNetworking/AFNetworking`.

[51] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *The 22nd USENIX Security Symposium (SECURITY)*, 2013.

[52] Z. Wang and A. Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[53] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios: Bring your own privacy & security to ios devices. In *8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*, May 2013.

[54] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning. Andbot: Towards advanced mobile botnets. In *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats*, LEET'11, 2011.

[55] M. Zheng, H. Xue, and T. Wei. Background monitoring on non-jailbroken ios 7 devices. `http://www.fireeye.com/blog/author/twei`.

A Appendices

A.1 Fairplay

The code segments of iOS apps on the App Store are encrypted with AES-128. Specifically, rather than using a single pair of key and IV (Initialization Vector) per app, each 4K bytes (i.e., memory page size) of the code segment of an app are encrypted with a unique pair of key and IV. These keys and IVs are also encrypted and stored in an `supp` file that is inside the `SC_Info` folder within the app archive. Upon loading an encrypted app into memory, iOS will derive the decryption keys and IVs from the `supp` file, the `sinf` file that is also inside the `SC_Info` folder, and the `sidb` file that resides

under `/private/var/mobile/Library/FairPlay/iTunes_Control/iTunes/`. A heavily obfuscated kernel module `FairPlayIOKit` and a heavily obfuscated user space daemon `fairplayd` are involved in this process.

Furthermore, this user space daemon `fairplayd` is also involved in the generation of `afsync.rq` and `afsync.rq.sig` mentioned in Section 2.2. After receiving the syncing request from a PC, the air traffic control service `atc` running in iOS devices will communicate with `fairplayd` through Mach messages to generate `afsync.rq` and `afsync.rq.sig`.

A.2 Measurement

Exclude small business networks. We plotted the cumulative distribution for number of distinct valid domains queried from all CIDs in a day in Figure 6. Some CIDs queried a lot more than 2,000 distinct domains in a day, e.g., 25,138,224. However, we only show in Figure 6 until 2,000 unique domains in the x-axis. The CDF curve grows extremely slowly after 1,000 unique domains, and 99.5% of CIDs queried fewer than 1,000 unique domains in a day. Therefore, we only use CIDs that queried fewer than 1,000 distinct valid domains per day in our experiment.

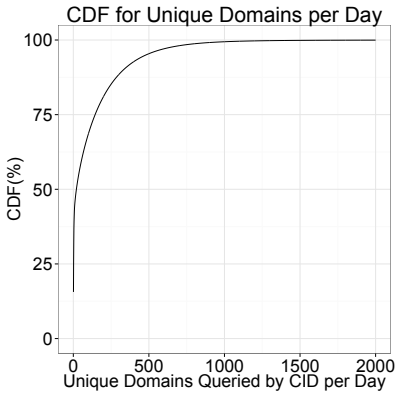


Figure 6: Cumulative distribution for number of distinct valid domains queried from all CIDs, on 10/12/2013.

A Large-Scale Analysis of the Security of Embedded Firmwares

Andrei Costin, Jonas Zaddach, Aurélien Francillon and Davide Balzarotti

EURECOM
Sophia Antipolis
France
{*name.surname*}@*eurecom.fr*

Abstract

As embedded systems are more than ever present in our society, their security is becoming an increasingly important issue. However, based on the results of many recent analyses of individual firmware images, embedded systems acquired a reputation of being insecure. Despite these facts, we still lack a global understanding of embedded systems' security as well as the tools and techniques needed to support such general claims.

In this paper we present the first public, large-scale analysis of firmware images. In particular, we unpacked 32 thousand firmware images into 1.7 million individual files, which we then statically analyzed. We leverage this large-scale analysis to bring new insights on the security of embedded devices and to underline and detail several important challenges that need to be addressed in future research. We also show the main benefits of looking at many different devices at the same time and of linking our results with other large-scale datasets such as the ZMap's HTTPS survey.

In summary, without performing sophisticated static analysis, we discovered a total of 38 previously unknown vulnerabilities in over 693 firmware images. Moreover, by correlating similar files inside apparently unrelated firmware images, we were able to extend some of those vulnerabilities to over 123 different products. We also confirmed that some of these vulnerabilities altogether are affecting at least 140*K* devices accessible over the Internet. It would not have been possible to achieve these results without an analysis at such wide scale.

We believe that this project, which we plan to provide as a firmware unpacking and analysis web service[1], will help shed some light on the security of embedded devices.

[1] `http://firmware.re`

1 Introduction

Embedded systems are omnipresent in our everyday life. For example, they are the core of various Common-Off-The-Shelf (COTS) devices such as printers, mobile phones, home routers, and computer components and peripherals. They are also present in many devices that are less consumer oriented such as video surveillance systems, medical implants, car elements, SCADA and PLC devices, and basically anything we normally call *electronics*. The emerging phenomenon of the Internet-of-Things (IoT) will make them even more widespread and interconnected.

All these systems run special software, often called *firmware*, which is usually distributed by vendors as *firmware images* or *firmware updates*. Several definitions for *firmware* exist in the literature. The term was originally introduced to describe the CPU microcode that existed "somewhere" between the hardware and the software layers. However, the word quickly assumed a broader meaning, and the IEEE Std 610.12-1990 [6] extended the definition to cover the *"combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device"*.

Nowadays, the term *firmware* is more generally used to describe the software that is embedded in a hardware device. Like traditional software, embedded devices' firmware may have bugs or misconfigurations that can result in vulnerabilities for the devices which run that particular code. Due to anecdotal evidence, embedded systems acquired a bad security reputation, generally based on case by case experiences of failures. For instance, a car model throttle control fails [47] or can be maliciously taken over [21, 55]; a home wireless router is found to have a backdoor [48, 7, 44], just to name a few recent examples. On the one hand, apart from a few works that targeted specific devices or software versions [39, 27, 63], to date there is still no large-scale security analysis of firmware images. On the other hand,

manual security analysis of firmware images yields very accurate results, but it is extremely slow and does not scale well for a large and heterogeneous dataset of firmware images. As useful as such individual reports are for a particular device or firmware version, these alone do not allow to establish a general judgment on the overall state of the security of firmware images. Even worse, the same vulnerability may be present in different devices, which are left vulnerable until those flaws are rediscovered independently by other researchers [48]. This is often the case when several *integration vendors* rely on the same subcontractors, tools, or SDKs provided by *development vendors*. Devices may also be branded under different names but may actually run either the same or similar firmware. Such devices will often be affected by exactly the same vulnerabilities, however, without a detailed knowledge of the internal relationships between those vendors, it is often impossible to identify such similarities. As a consequence, some devices will often be left affected by known vulnerabilities even if an updated firmware is available.

1.1 Methodology

Performing a large-scale study of the security of embedded devices by actually running the physical devices (i.e., using a dynamic analysis approach) has several major drawbacks. First of all, physically acquiring thousands of devices to study would be prohibitively expensive. Moreover, some of them may be hard to operate outside the system for which they are designed — e.g., a throttle control outside a car. Another option is to analyze existing online devices as presented by Cui and Stolfo [29]. However, some vulnerabilities are hard to find by just looking at the running device, and it is ethically questionable to perform any nontrivial analysis on an online system without authorization.

Unsurprisingly, static analysis scales better than dynamic analysis as it does not require access to the physical devices. Hence, we decided to follow this approach in our study. Our methodology consists of collecting firmware images for as many devices and vendors as possible. This task is complicated by the fact that firmware images are diverse and it is often difficult to tell firmware images apart from other files. In particular, distribution channels, packaging formats, installation procedures, and availability of meta-data often depend on the vendor and on the device type. We then designed and implemented a distributed architecture to unpack and run simple static analysis tasks on the collected firmware images. However, the contribution of this paper is not in the static analysis techniques we use (for example, we did not perform any static *code* analysis), but to show the advantages of an horizontal, large-scale exploration. For this reason, we implemented a correlation engine to compare and find similarities between all the objects in our dataset. This allowed us to quickly "propagate" vulnerabilities from known vulnerable devices to other systems that were previously not known to be affected by the same vulnerability.

Most of the steps performed by our system are conceptually simple and could be easily performed manually on a few devices. However, we identified *five major challenges* that researchers need to address in order to perform large scale experiments on thousands of different firmware images. These include the problem of building a representative dataset (Challenge A in Section 2), of properly identifying individual firmware images (Challenge B in Section 2), of unpacking custom archive formats (Challenge C in Section 2), of limiting the required computation resources (Challenge D in Section 2), and finally of finding an automated way to confirm the results of the analysis (Challenge E in Section 2). While in this paper we do not propose a complete solution for all these challenges, we discuss the way and the extent to which we dealt with some of these challenges to perform a systematic, automated, large-scale analysis of firmware images.

1.2 Results Overview

For our experiments we collected an initial set of `759,273` files (totaling `1.8TB` of storage space) from publicly accessible firmware update sites. After filtering out the obvious noise, we were left with `172,751` potential firmware images. We then sampled a set of `32,356` firmware candidates that we analyzed using a private cloud deployment of `90` worker nodes. The analysis and reports resulted in a `10GB` database.

The analysis of sampled files led us to automatically discover and report 38 new vulnerabilities (fixes for some of these are still pending) and to confirm several that were already known [44, 48]. Some of our findings include:

- We extracted private RSA keys and their self-signed certificates used in about 35,000 online devices (mainly associated with surveillance cameras).
- We extracted several dozens of hard-coded password hashes. Most of them were weak, and therefore we were able to easily recover the original passwords.
- We identified a number of possible backdoors such as the `authorized_keys` file (which lists the SSH keys that are allowed to remotely connect to the system), a number of hard-coded `telnetd` credentials affecting at least 2K devices, hard-coded web-login admin credentials affecting at least 101K de-

vices, and a number of backdoored daemons and web pages in the web-interface of the devices.
- Whenever a new vulnerability was discovered (by other researchers or by us) our analysis infrastructure allowed us to quickly find related devices or firmware versions that were likely affected by the same vulnerability. For example, our *correlation techniques* allowed us to correctly extend the list of affected devices for variations of a `telnetd` hard-coded credentials vulnerability. In other cases, this led us to find a vulnerability's root problem spread across multiple vendors.

1.3 Contributions

In summary this paper makes the following contributions:

- We show the advantages of performing a large-scale analysis of firmware images and describe the main challenges associated with this activity.
- We propose a framework to perform firmware collection, filtering, unpacking and analysis at large scale.
- We implemented several efficient static techniques that we ran on 32,356 firmware candidates.
- We present a correlation technique which allows to propagate vulnerability information to similar firmware images.
- We discovered 693 firmware images affected by at least one vulnerability and reported 38 new CVEs.

2 Challenges

As mentioned in the previous section, there are clear advantages of performing a wide-scale analysis of embedded firmware images. In fact, as is often the case in system security, certain phenomena can only be observed by looking at the global picture and not by studying a single device (or a single family of devices) at a time.

However, large-scale experiments require automated techniques to obtain firmware images, unpack them, and analyze the extracted files. While these are easy tasks for a human, they become challenging when they need to be fully automated. In this section we summarize the five main challenges that we faced during the design and implementation of our experiments.

Challenge A: Building a Representative Dataset

The embedded systems environment is heterogeneous, spanning a variety of devices, vendors, architectures, instruction sets, operating systems, and custom components. This makes the task of compiling a *representative* and *balanced* dataset of firmware images a difficult problem to solve.

The real market distribution of a certain hardware architecture is often unknown, and it is hard to compare different classes of devices (e.g., medical implants vs. surveillance cameras). Which of them need to be taken into account to build a representative firmware dataset? How easy is it to generalize a technique that has only been tested on a certain brand of routers to other vendors? How easy is it to apply the same technique to other classes of devices such as TVs, cameras, insulin pumps, or power plant controllers?

From a practical point of view, the lack of centralized points of collection (such as the ones provided by antivirus vendors or public sandboxes in the malware analysis field) makes it difficult for researchers to gather a large and well triaged dataset. Firmware often needs to be downloaded from the vendor web pages, and it is not always simple, even for a human, to tell whether or not two firmware images are for the same physical device.

Challenge B: Firmware Identification

One challenge often encountered in firmware analysis and reverse engineering is the difficulty of reliably extracting meta-data from a firmware image. For instance, such meta-data includes the vendor, the device product code and purpose, the firmware version, and the processor architecture, among many other details.

In practice, the diversity of firmware file formats makes it harder to even recognize that a given file downloaded from a vendor website is a firmware at all. Often firmware updates come in unexpected formats such as *HP Printer Job Language* and *PostScript* documents for printers [24, 23, 27], *DOS executables* for BIOS, and *ISO images* for hard disk drives [72].

In many cases, the only source of reliable information is the official vendor documentation. While this is not a problem when looking manually at a few devices, extending the analysis to hundreds of vendors and thousands of firmware images automatically downloaded from the Internet is challenging. In fact, the information retrieval process is hard to automate and is error prone, in particular for certain classes of meta-data. For instance, we often found it hard to infer the correct version number. This makes it difficult for a large-scale collection and analysis system to tell which is the latest version available for a certain device, and even if two firmware images corresponded to different versions for the same device. This further complicates the task of building an unbiased dataset.

Challenge C: Unpacking and Custom Formats

Assuming the analyst succeeded in collecting a representative and well labeled dataset of firmware images, the next challenge consists in locating and extracting important functional blocks (e.g., binary code, configuration files, scripts, web interfaces) on which static analysis routines can be performed.

While this task would be easy to address for traditional software components, where standardized formats for the distribution of machine code (e.g., PE and ELF), resources (e.g., JPEG and GZIP) and groups of files (e.g., ZIP and TAR) exist, embedded software distribution lacks standards. Vendors have developed their own file formats to describe flash and memory images. In some cases those formats are compressed with non-standard compression algorithms. In other cases those formats are obfuscated or encrypted to prevent analysis. Monolithic firmware, in which the bootloader, the operating system kernel, the applications, and other resources are combined together in a single memory image are especially challenging to unpack.

Forensic strategies, like file *carving*, can help to extract known file formats from a binary blob. Unfortunately those methods have drawbacks: On the one hand, they are often too aggressive with the result of extracting data that matches a file pattern only by chance. On the other hand, they are computationally expensive, since each unpacker has to be tried for each file offset of the binary firmware blob.

Finally, if a binary file has been extracted that does not match any known file pattern, it is impossible to say if this file is a data file, or just another container format that is not recognized by the unpacker. In general, we tried to unpack at least until reaching uncompressed files. In some cases, our extraction goes one step further and tries to extract sections, resources and compressed streams (e.g., for the ELF file format).

Challenge D: Scalability and Computational Limits

One of the main advantages of performing a wide-scale analysis is the ability of correlating information across multiple devices. For example, this allowed us to automatically identify the re-use of vulnerable components among different firmware images, even from different vendors.

Capturing the global picture of the relationship between firmware images would require the one-to-one comparison of each pair of unpacked files. Fuzzy hashes (such as sdhash [62] and ssdeep [54]) are a common and effective solution for this type of task and they have been successfully used in similar domains, e.g., to correlate samples that belong to the same malware families [35, 15]. However, as described in more detail in Section 3.4, computing the similarity between the objects extracted from 26,275 firmware images requires 10^{12} comparisons. Using the simpler fuzzy hash variant, we estimate that on a single dual-core computer this task would take approximately 850 days[2]. This simple estimation highlights one of the possible *computational challenges* associated with a large-scale firmware analysis. Even if we had a perfect database design and a highly optimized in-memory database, it would still be hard to compute, store, and query the fuzzy hash scores of all pairs of unpacked files. A distributed computational infrastructure can help reduce the total time since the task itself is parallelizable [57]. However, since the number of comparisons grows quadratically with the number of elements to compare, this problem quickly becomes impracticable for large image datasets. If, for example, one would like to build a fuzzy hash database for our whole dataset, which is just five times the size of the current sampled dataset, this effort would already take more than 150 CPU years instead of 850 CPU days. Our attempt to use the GPU-assisted fuzzy hashing provided by `sdhash` [62] only resulted in a limited speedup that was not sufficient to perform a full-scale comparison of all files in our dataset.

Challenge E: Results Confirmation

The first four challenges were mostly related to the collection of the dataset and the pre-processing of the firmware images. Once the code or the resources used by the embedded device have been successfully extracted and identified, researchers can focus their attention on the static analysis. Even though the details and goals of this step are beyond the scope of this paper, in Section 3.3 we present some examples of simple static analysis and we discuss the advantages of performing these techniques on a large scale.

However, one important research challenge remains regarding the way the results of static analysis can be confirmed. For example, we can consider a scenario where a researcher applies a new vulnerability detection technique to several thousand firmware images. Those images were designed to run on specific embedded devices, most of which are not available to the researcher and would be hard and costly to acquire. Lacking the proper hardware platform, there is still no way to manually or automatically test the *affected code* to confirm or deny the findings of the static analysis.

For example, in our experiments we identified a firmware image that included the `PHP 5.2.12` banner string. This allowed us to easily identify several vulnerabilities

[2] This is mainly because comparing fuzzy hashes is not a simple bit string comparison but actually involves a rather complex algorithm and high computational effort.

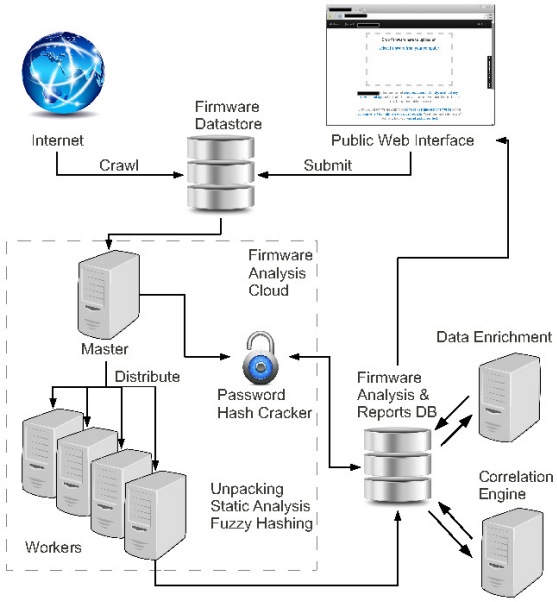


Figure 1: Architecture of the entire system.

associated with that version of the PHP interpreter. However, this is insufficient to determine if the PHP interpreter is vulnerable, since the vendor may have applied patches to correct known vulnerabilities without this being reflected in the version string. In addition, the vendor might have used an architecture and/or a set of compilation options which produced a non-vulnerable build of the component. Unfortunately, even if a proof of concept attack exists for that vulnerability, without the proper hardware it is impossible to test the firmware and confirm or deny the presence of the problem.

Confirming the results of the static analysis on firmware devices is a tedious task requiring manual intervention from an expert. Scaling this effort to thousands of firmware images is even harder. Therefore, we believe the development of new techniques is required to accurately deal with this problem at a large scale.

3 Setup

In this section we first present the design of our distributed static analysis and correlation system. Then we detail the techniques we used, and how we addressed the challenges described in Section 2.

3.1 Architecture

Figure 1 presents an overview of our architecture. The first component of our analysis platform is the *firmware data store*, which stores the unmodified firmware files that have been retrieved either by the *web crawler* or that have been submitted through the public *web interface*. When a new file is received by the firmware data store, it is automatically scheduled to be processed by the *analysis cloud*. The analysis cloud consists of a master node, and a number of worker and hash cracking nodes. The *master node* distributes unpacking jobs to the *worker nodes* (Figure 2), which unpack and analyze firmware images. *Hash cracking nodes* process password hashes that have been found during the analysis, and try to find the corresponding plaintext passwords. Apart from coordinating the worker nodes, the master node also runs the *correlation engine* and the *data enrichment system* modules. These modules improve the reports with results from the cross-firmware analysis.

The analysis cloud is where the actual analysis of the firmware takes place. Each firmware image is first submitted to the *master node*. Subsequently, *worker nodes* are responsible for unpacking and analyzing the firmware and for returning the results of the analysis back to the master node. At this point, the master node will submit this information to the *reports database*. If there were any uncracked password hashes in the analyzed firmware, it will additionally submit those hashes to one of the *hash cracking nodes* which will try to recover the plaintext passwords.

It is important to note that only the results of the analysis and the meta-data of the unpacked files are stored in the database. Even though we do not currently use the extracted files after the analysis, we still archive them for future work, or in case we want to review or enhance a specific set of analyzed firmware images.

The architecture contains two other components: the *correlation engine* and the *data enrichment system*. Both of them fetch the results of the firmware analysis from the reports database and perform additional tasks. The correlation engine identifies a number of “interesting” files and tries to correlate them with any other file present in the database. The enrichment system is responsible for enhancing the information about each firmware image by performing online scans and lookup queries (e.g., detecting vendor name, device name/code and device category).

In the remainder of this section we describe each step of the firmware analysis in more detail so that our experiments can be reproduced.

3.2 Firmware Acquisition and Storage

The first step of our experiments consisted in gathering a firmware collection for analysis. We achieved this goal by using mainly two methods: a web crawler that automatically downloads files from manufacturers’ websites and specialized mirror sites, and a website with a submis-

sion interface where users can submit firmware images for analysis.

We initialized the crawler with tens of support pages from well known manufacturers such as Xerox, Bosch, Philips, D-Link, Samsung, LG, Belkin, etc. Second, we used public FTP indexing engines [3] to search for files with keywords related to firmware images (e.g., `firmware`). The result of such searches yields either directory URLs, which are added to the crawler list of URLs to index and download, or file URLs, which are directly downloaded by the crawler. At the same time, the script strips filenames out of the URLs to create additional directory URLs.

Finally, we used Google Custom Search Engines (GCSE) [3] to create customized search engines. GCSE provides a flexible API to perform advanced search queries and returns results in a structured way. It also allows to programmatically create a very customized CSE on-the-fly using a combination of RESTful and XML APIs. For example, a CSE is created using `support.nikonusa.com` as the "Sites to Search" parameter. Then a firmware related query is used on the CSE such as `''firmware download''`. The CSE from the above example returns 2,210 results at the time of this publication. The result URLs along with associated meta-data are retrieved via the JSON API. Each URL was then used by the crawler or as part of other dynamic CSE, as previously described. This allowed us to mine additional firmware images and firmware repositories.

We chose not to filter data at collection time, but to download files greedily, deciding at a later stage if the collected files were firmware images or not. The reason for this decision is two-fold. First, accompanying files such as manuals and user guides can be useful for finding additional download locations or for extracting contained information (e.g., model, default passwords, update URLs). Second, as we mentioned previously, it is often difficult to distinguish firmware images from other files. For this reason, filtering a large dataset is better than taking a chance to miss firmware files during the downloading phase. In total, we crawled 284 sites and stopped downloading once the collection of files reached 1.8TB of storage. The actual storage required for this amount of data is at least 3-4 times larger, since we used mirrored backup storage, as well as space for keeping the unpacked files and files generated during the unpacking (e.g., logs and analysis results).

The public *web submission interface* provides a means for security researchers to submit firmware files for analysis. After the analysis is completed, the platform produces a report with information about the firmware contents as well as similarities to other firmware in our database. We have already received tens of firmware images through the submission interface. While this is currently a marginal source of firmware files, we expect that more firmware will be submitted as we advertise our service. This will also be a unique chance to have access to firmware images that are not generally available and, for example, need to be manually extracted from a device.

Files fetched by the web crawler and received from the web submission interface are added to the *firmware data store*. Files are simply stored on a file system and a database is used for meta-data (e.g., file checksum, size, download location).

[3] FTP indexing engines such as: `www.mmnt.ru`, `www.filemare.com`, `www.filewatcher.com`, `www.filesearching.com`, `www.ftpsearch.net`, `www.search-ftps.com`

3.3 Unpacking and Analysis

The next step towards the analysis of a firmware image is to unpack and extract the contained files or objects. The output of this phase largely depends on the type of firmware. In some examples, executable code and resources (such as graphics files or HTML code) can be linked into a binary blob that is designed to be directly copied into memory by a bootloader and then executed. Some other firmware images are distributed in a compressed and obfuscated file which contains a block-by-block copy of a flash image. Such an image may consist of several partitions containing a bootloader, a kernel and a file system.

Unpacking Frameworks

There are three main tools to unpack arbitrary firmware images: binwalk [41], FRAK [26] and Binary Analysis Toolkit (BAT) [66].

Binwalk is a well known firmware unpacking tool developed by Craig Heffner [41]. It uses pattern matching to locate and carve files from a binary blob. Additionally, it also extracts meta-data such as license strings.

FRAK is an unpacking toolkit first presented by Cui et al. [27]. Even though the authors mention that the tool would be made publicly available, we were not able to obtain a copy. We therefore had to evaluate its unpacking performance based on the device vendors and models that FRAK supports, according to [27]. We estimated that FRAK would have unpacked less than 1% of the files we analyzed, while our platform was able to unpack more than 81% of them. This said, both would be complementary as some of the file formats FRAK unpacks are unsupported by our tool at present.

The Binary Analysis Toolkit (BAT), formerly known as GPLtool, was originally designed by Tjaldur software to detect GPL violations [45, 66]. To this end, it recursively extracts files from a firmware blob and matches strings with a database of known strings from

Table 1: Comparison of Binwalk, BAT, FRAK and our framework. The last three columns show if the respective unpacker was able to extract the firmware. Note that this is a non statistically significant sample which is given for illustrating unpacking performance (manual analysis of each firmware is time consuming). As FRAK was not available for testing, its unpacking performance was estimated based on information from [26]. The additional performance of our framework stems from the many customizations we have incrementally developed over BAT (Figure 2).

Device	Vendor	OS	Binwalk	BAT	FRAK	Our framework
PC	Intel	BIOS	✗	✗	✗	✗
Camera	STL	Linux	✗	✓	✗	✓
Router	Bintec	-	✗	✗	✗	✗
ADSL Gateway	Zyxel	ZynOS	✓	✓	✗	✓
PLC	Siemens	-	✓	✓	✗	✓
DSLAM	-	-	✓	✓	✗	✓
PC	Intel	BIOS	✓	✓	✗	✓
ISDN Server	Planet	-	✓	✓	✗	✓
Voip	Asotel	Vxworks	✓	✓	✗	✓
Modem	-	-	✗	✗	✗	✓
Home Automation	Belkin	Linux	✗	✗	✗	✓
			55%	64%	0%	82%

GPL projects. Additionally, BAT supports file carving similar to binwalk.

Table 1 shows a simple comparison of the unpacking performance of each framework on a few samples of firmware images. We chose to use BAT because it is the most complete tool available for our purposes. It also has a significantly lower rate of false positive extractions compared to binwalk. In addition, binwalk did not support recursive unpacking at the time when we decided on an unpacking framework. Nevertheless, the interface between our framework and BAT has been designed to be generic so that integrating other unpacking toolkits (such as binwalk) is easy.

We developed a range of additional plugins for BAT. These include plugins which extract interesting strings (e.g., software versions or password hashes), add unpacking methods, gather statistics and collect interesting files such as private key files or `authorized_keys` files. In total we added 35 plugins to the existing framework.

Password Hash Cracking

Password hashes found during the analysis phase are passed to a hash cracking node. These nodes are dedicated physical hosts with a Nvidia Tesla GPU [56] that run a CUDA-enabled [59] version of *John The Ripper* [60]. John The Ripper is capable of brute forcing most encoded password hashes and detecting the type of hash and salt used. In addition to this, a dictionary can be provided to seed the password cracking. For each brute force attempt, we provide a dictionary built from common password lists and strings extracted from firmwares, manuals, *readme* files and other resources. This allows to find both passwords that are directly present in those files as well as passwords that are weak and based on keywords related to the product.

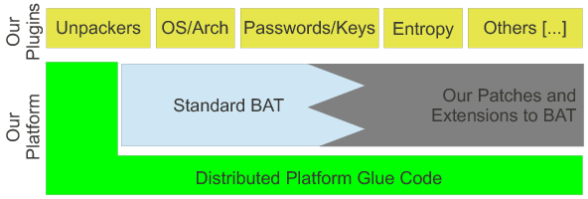


Figure 2: Architecture of a single worker node.

Parallelizing the Unpacking and Analysis

To accelerate the unpacking process, we distributed this task on several worker nodes. Our distributed environment is based on the *distributed-python-for-scripting* framework [65]. Data is synchronized between the repository and the nodes using *rsync (over ssh)* [67].

Our loosely coupled architecture allows us to run worker nodes virtually anywhere. For instance, we instantiated worker virtual machines on a local VMware server and several OpenStack servers, as well as on Amazon EC2 instances. At the time of this publication we were using 90 such virtual machines to analyze firmware files.

3.4 Correlation Engine

The unpacked firmware images and analysis results are stored into the *analysis & reports database*. This allows us to perform queries, to generate reports and statistics, and to easily integrate our results with other external components. The correlation engine is designed to find similarities between different firmware images. In particular, the comparison is made along four different dimensions: shared credentials, shared self-signed certificates, common keywords, and fuzzy hashes of the firmwares and objects within the firmwares.

Shared Credentials and Self-Signed Certificates

Shared credentials (such as hard coded *non-trivial* passwords) and shared self-signed certificates are effective in finding strong connections between different firmware images of the same vendor, or even firmwares of different vendors. For example, we were able to correlate two brands of CCTV systems based on a common *non-trivial* default password.

Therefore, finding a password of one vendor's product can directly impact the security of others. We also found a similar type of correlation for two other CCTV vendors that we linked through the same self-signed certificate, as explained in Section 5.2.

Keywords

Keywords correlation is based on specific strings extracted by our static analysis plugins. In some cases, for example in Section 5.1, the keyword "`backdoor`" revealed several other keywords. By using the extended set of keywords we clustered several vendors prone to the same backdoor functionality, possibly affecting $500,000$ devices. In other cases, files inside firmware images contain compilation and SDK paths. This turns out to be sufficient to cluster firmware images of different devices.

Fuzzy hashes

Fuzzy hash triage (comparison, correlation and clustering) is the most generic correlation technique used by our framework. The engine computes both the *ssdeep* and the *sdhash* of every single object extracted from the firmware image during the unpacking phase. This is a powerful technique that allows us to find files that are "similar" but for which a traditional hash (such as *MD5* or *SHA1*) would not match. Unfortunately, as we already mentioned in Section 2, a complete one-to-one comparison of fuzzy hashes is currently infeasible on a large scale. Therefore, we compute the fuzzy hashes of each file that was successfully extracted from a firmware image and store this result. When a file is found to be interesting we perform the fuzzy hash comparison between this file's hash and all stored hashes.

For example, a file (or all files unpacked from a firmware) may be flagged as interesting because it is affected by a known vulnerability, or because we found it to be vulnerable by static analysis. If another firmware contains a file that is similar to a file from a vulnerable firmware, then there might be a chance that the first firmware is also vulnerable. We present such an example in Section 5.3, where this approach was successful and allowed us to propagate known vulnerabilities of one device to other similar devices of *different* vendors.

Future work

In the literature, there are several approaches proposed to perform comparison, clustering, and triage on a large scale. Jang et al. propose large-scale triage techniques of PC malware in BitShred [52]. The authors concluded that at the rate of 8,000 unique malware samples per day, which required 31M comparisons, it is unfeasible on a single CPU to perform one-to-one comparisons to find malware families using hierarchical clustering. French and Casey [13] propose, before fuzzy hash comparison, to perform a "bins" partitioning approach based on the block and file sizes. This approach, for their particular dataset and bins partitioning strategy, allowed on average to reduce the search space for a given fuzzy hash down to 16.9%. Chakradeo et al. [20] propose MAST, an effective and well performing triage architecture for mobile market applications. It solves the manual and resource-intensive automated analysis at market-scale using Multiple Correspondence Analysis (MCA) statistical method.

As a future work, there are several possible improvements to our approach. For instance, instead of performing all comparisons on a single machine, we could adopt a distributed comparison and clustering infrastructure, such as the Hadoop implementation of MapReduce [32] used by BitShred. Second, on each comparison and clustering node we could use the "bins" partitioning approach from French and Casey [13].

3.5 Data Enrichment

The data enrichment phase is responsible for extending the knowledge base about firmware images, for example by performing automated queries and passive scans over the Internet. In the current prototype, the data enrichment relies on two simple techniques. First, it uses the `<title>` tag of web pages and `authentication realms` of web servers when these are detected inside a firmware. This information is then used to build targeted *search queries* (such as *"intitle:Router ABC-123 Admin Page"*) for both Shodan [5] and GCSE.

Second, we correlate SSL certificates extracted from firmware images to those collected by the ZMap project. ZMap was used in [37] to scan the whole IPv4 address space on the 443 port, collecting SSL certificates in a large database.

Correlating these two large-scale databases (i.e., ZMap's HTTPS survey and our firmware database) provides new insights. For example, we are able to quickly evaluate the severity of a particular vulnerability by identifying publicly reachable devices that are running a given firmware image. This gives a good estimate for the number of publicly accessible vulnerable devices.

For instance, our framework found 41 certificates having unprotected private keys. Those keys were extracted from firmware images in the unpacking and analysis phase. The data enrichment engine subsequently found the same self-signed certificate in over 35K devices reachable on the Internet. We detail this case study in Section 5.2.

3.6 Setup Development Effort

Our framework relies on many existing tools. In addition to this, we have put a considerable effort (over 20k lines of code according to `sloccount` [68]) to extend BAT, develop new unpackers, create the results analysis platform and run results interpretation.

4 Dataset and Results

In this section we describe our dataset and we present the results of the global analysis, including the discussion of the new vulnerabilities and the common bad practices we discovered in our experiments. In Section 5, we will then present a few concrete case studies, illustrating how such a large dataset can provide new insights into the security of embedded systems.

4.1 General Dataset Statistics

While we currently collect firmware images from multiple sources, most of the images in our dataset have been downloaded by crawling the Internet. As a consequence, our dataset is biased towards devices for which firmware updates can be found online, and towards known vendors that maintain well organized websites.

We also decided to exclude firmware images of smartphones from our study. In fact, popular smartphone firmware images are complete operating system distributions, most of them iOS, Android or Windows based – making them closer to general purpose systems than to embedded devices.

Our crawler collected 759,273 files, for a total of 1.8TB of data. After filtering out the files that were clearly unrelated (e.g., manuals, user guides, web pages, empty files) we obtained a dataset of 172,751 files. Our architecture is constantly running to fetch more samples and analyze them in a distributed fashion. At the time of this publication the system was able to process (unpack and analyze) 32,356 firmware images.

Firmware Identification The problem of properly identifying a firmware image (Challenge 2) still requires a considerable amount of manual effort. Doing so accurately and automatically at a large scale is a daunting task. Nevertheless, we are interested in having an estimate of the number of actual firmware images in our dataset.

For this purpose we manually analyzed a number of random samples from our dataset of 172,751 potential firmware images and computed a *confidence interval* [19] to estimate the global representativeness in the dataset. In particular, after manually analyzing 130 random files from the total of 172,751, we were able to mark only 44 as firmware images. This translates to a proportion of 34% ($\pm$ 8%) firmware images on our dataset – with a 95% confidence. The manual analysis process took approximately one person-week because the inspection of the extracted files for firmware code is quite tedious.

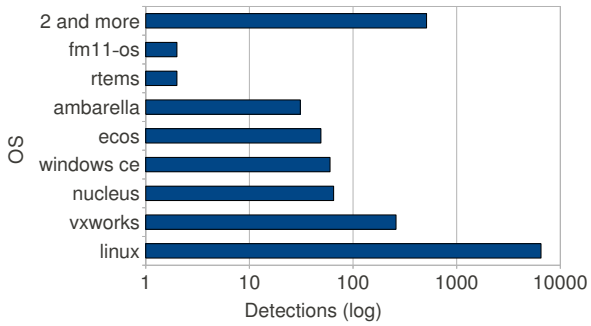


Figure 3: OS distribution among firmware images.

We can therefore expect our dataset to contain between 44,431 and 72,520 firmware images (by applying 34%−8%, and 34%+8% respectively, to the entire candidates set of 172,751). While the range is still relatively large, this estimation gives a 95% reliable measure of the useful data in our sample. We also developed a heuristic to automatically detect if a file is successfully unpacked or not. This heuristic takes multiple parameters, such as the number, type and size of files carved out from a firmware, into account. Such an empirical heuristic is not perfect, but it can guide our framework to mark a file as unpacked or not, and then take actions accordingly.

Files Analysis As described in Section 3.3, unpacking unknown files is an error-prone and time-consuming task. In fact, when the file format is not recognized, unpacking relies on a slow and imprecise carving approach. File carving is essentially an attempt to unpack at every offset of the file, iterating over several known signatures (e.g., archive magic headers).

As a result, out of the 32,356 files we processed so far, 26,275 were successfully unpacked. The process is nevertheless continuous and more firmware images are being unpacked over time.

4.2 Results Overview

In the rest of the section we present the results of the analysis performed by our plugins right after each firmware image was unpacked.

Files Formats The majority of initial files being unpacked were identified as *compressed files* or *raw data*. Once unpacked, most of those firmware images were

identified as targeting ARM (63%) devices, followed by MIPS (7%). As reported in Figure 3, Linux is the most frequently encountered embedded operating system in our dataset – being present in more than three quarters (86%) of all analyzed firmware images. The remaining images contain proprietary operating systems like VxWorks, Nucleus RTOS and Windows CE, which altogether represent around 7%. Among Linux based firmware images, we identified 112 distinct Linux kernel versions.

Password Hashes Statistics Files like `/etc/passwd` and `/etc/shadow` store hashed versions of account credentials. These are usual targets for attackers since they can be used to retrieve passwords which often allow to login remotely to a device at a later time. Hence, an analysis of these files can help understanding how well an embedded device is protected.

Our plugin responsible for collecting entries from `/etc/passwd` and `/etc/shadow` files retrieved 100 distinct password hashes, covering 681 distinct firmware images and belonging to 27 vendors. We were also able to recover the plaintext passwords for 58 of those hashes, which occur in 538 distinct firmware images. The most popular passwords were `<empty>`, `pass`, `logout`, and `helpme`. While these may look trivial, it is important to stress that they are actually used in a large number of embedded devices.

Certificates and Private RSA Keys Statistics Many vendors include self-signed certificates inside their firmware images [43, 42]. Due to bad practices in both *release management* and *software design*, some vendors also include the private keys (e.g., PEM, GPG), as confirmed by recent advisories [49, 51].

We developed two simple plugins for our system which collect SSL certificates and private keys. These plugins also collect their fingerprints and check for empty or trivial passphrases. So far, we have been able to extract 109 private RSA keys from 428 firmware images and 56 self-signed SSL certificates out of 344 firmware images. In total, we obtained 41 self-signed SSL certificates together with their corresponding private RSA keys. By looking up those certificates in the public ZMap datasets [36], we were able to automatically locate about 35,000 active online devices.

For all these devices, if the certificate and private key are not regenerated on the first boot after a firmware update, HTTPS encryption can be easily decrypted by an attacker by simply downloading a copy of the firmware image. In addition, if both a regenerated and a firmware-shipped self-signed certificate are used interchangeably, the user of the device may still be vulnerable to man-in-the-middle (MITM) attacks.

Packaging Outdated and Vulnerable Software Another interesting finding relates to bad *release management* by embedded firmware vendors. Firmware images often rely on many third-party software and libraries. Those keep updating and have security fixes every now and then. OWASP Top Ten [61] lists *"Using Components with Known Vulnerabilities"* at position nine and underlines that *"upgrading to these new versions is critical"*.

In one particular case, we identified a relatively recently released firmware image that contained a kernel (version 2.4.20) that was built and packaged ten years after its initial release. In another case, we discovered that some recently released firmware images contained nine years old BusyBox versions.

Building Images as *root* While prototyping, putting together a build environment as fast as possible is very important. Unfortunately, sometimes the easiest solution is just to setup and run the entire toolchains as superuser.

Our analysis plugins extracted several compilation banners such as `Linux version 2.6.31.8-mv78100 (root@ubuntu) (gcc version 4.2.0 20070413 (prerelease)) Mon Nov 7 16:51:58 JST 2011` or `BusyBox v1.7.0 (2007-10-15 19:49:46 IST)`.

24% of the 450 unique banners we collected containing the `user@host` combinations were associated to the `root` user. In addition to this, among the 267 unique hostnames extracted from those banners, *ten* resolved to public IP addresses and *one* of these even accepted incoming SSH connections.

All these findings reveal a number of unsafe practices ranging from *build management* (e.g., build process done as `root`) to *infrastructure management* (e.g., build hosts reachable over public networks), to *release management* (e.g., usernames and hostnames not removed from production release builds).

Web Servers Configuration We developed plugins to analyze the configuration files of web servers embedded in the firmware images such as `lighttpd.conf` or `boa.conf`. We then parsed the extracted files to retrieve specific configuration settings such as the running user, the documents root directory, and the file containing authentication secrets. We collected in total 847 distinct web server configuration files and the findings were discouraging. We found that in more than 81% of the cases the web servers were configured to run as a privileged user (i.e., having a setting such as `user=root`). This reveals unsafe practices of insecure design and configuration. Running the web server of an embedded device with unnecessarily high privileges can be extremely risky since the security of the entire device can be compromised by finding a vulnerability in one of the web components.

5 Case Studies

5.1 Backdoors in Plain Sight

Many backdoors in embedded systems have been reported recently, ranging from very simple cases [44] to others that were more difficult to discover [50, 64]. In one famous case [44], the backdoor was found to be activated by the string "`xmlset_roodkcableoj28840ybtide`" (i.e., `edit by 04882 joel backdoor` in reverse). This fully functional backdoor was affecting three vendors. Interestingly enough, this backdoor may have been detected earlier by a simple keyword matching on the open source release from the vendor[2].

Inspired by this case, we performed a string search in our dataset with various backdoor related keywords. Surprisingly, we found 1198 matches, in 326 firmware candidates.

Among those search results, several matched the firmware of a home automation device from a major vendor. According to download statistics from Google Play and Apple App Store, more than half a million users have downloaded an app for this device [9, 8].

We manually analyzed the firmware of this Linux-based embedded system and found that a daemon process listens on a network multicast address. This service allows execution of remote commands with `root` privileges without any authentication to anybody in the local network. An attacker can easily gain full control if he can send multicast packets to the device.

We then used this example as a *seed* for our *correlation engine*. With this approach we found exactly the same backdoor in two other classes of devices from two different vendors. One of them was affecting 109 firmware images of 44 camera models of a major CCTV solutions vendor, *Vendor C*. The other case is affecting three firmware images for home routers of a major networking equipment vendor, *Vendor D*.

We investigated the issue and found that the affected devices were relying on the same provider of a System on a Chip (SoC) for networking devices. It seems that this backdoor is intended for system debugging, and is part of a development kit. Unfortunately we were not able to locate the source of this binary. We plan to acquire some of those devices to verify the exploitability of the backdoor.

5.2 Private SSL Keys

In addition to the backdoors left in firmware images from *Vendor C*, we also found many firmware images containing public and *private RSA key* pairs. Those unprotected keys are used to provide SSL access to the CCTV camera's web interface. Surprisingly, this private key is the same across many firmware images of the same brand.

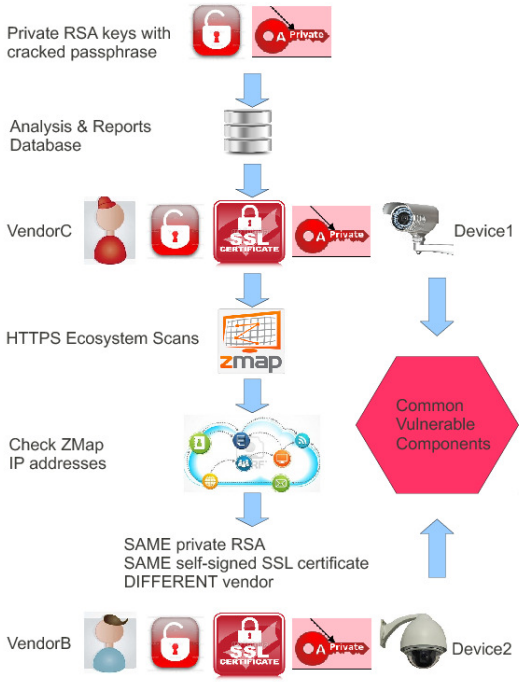


Figure 4: Correlation engine and shared self-signed certificates clustering.

Our platform automatically extracts the fingerprint of the public keys, private keys and SSL certificates. Those keys are then searched in ZMap's HTTPS survey database [36, 37]. *Vendor C*'s SSL certificate was found to be used by around 30K online IP addresses, most likely each corresponding to a single online device. We then fetched the web pages available at those addresses (without trying to authenticate). Surprisingly, we found CCTV cameras branded by another vendor – *Vendor B* – which appears to be an *integrator*. Upon inspection, cameras of *Vendor B* served *exactly the same* SSL certificate as cameras from *Vendor C* (including the SSL *Common Name*, and SSL *Organizational Unit* as well as many other fields of the SSL certificate). The only difference is that CCTV cameras of *Vendor B* returned branded authentication realms, error messages and logos. The *correlation engine* findings are summarized in Figure 4.

Unfortunately, the firmware images from *Vendor B* do not seem to be publicly available. We are planning to obtain a device to extract its firmware and to confirm our findings. We have reported these issues to the vendor. Nevertheless, it is very likely that devices from *Vendor B* are also vulnerable to the multicast packet backdoor given the clear relationship with *Vendor C* that that our platform discovered.

5.3 XSS in WiFi Enabled SD Cards?

SD cards are often more complex than one would imagine. Most SD cards actually contain a processor which runs firmware. This processor often manages functions such as the flash memory translation layer and wear leveling. Security issues have been previously shown on such SD cards [69].

Some SD cards have an embedded WiFi interface with a full fledged web server. This interface allows direct access to the files on the SD card without ejecting it from the device in which it is inserted. It also allows administration of the SD card configuration (e.g., WiFi access points).

We manually found a Cross Site Scripting (XSS) vulnerability in one of these web interfaces, which consists of a `perl` based web application. As this web application does not have platform specific binary bindings, we were able to load the files inside a similar Boa web server on a PC and confirm the vulnerability.

Once we found the exact `perl` files responsible for the XSS, we used our correlation engine based on fuzzy hashes. With this we automatically found another SD card firmware that is vulnerable to the same XSS. Even though the `perl` files were slightly different, they were clearly identified as similar by the fuzzy hash. This correlation would not have been detected by a normal checksum or by a regular hash function.

The process is visualized in Figure 5. The file (*) was found vulnerable. Subsequently, we identified correlated files based on fuzzy hashing. Some of them were related to the same firmware or a previous version of the firmware of the *same* vendor (in red). Also, fuzzy hash correlation identified a similar file in a firmware from a *different* vendor (in orange) that is vulnerable to the same weakness. It further identified some non-vulnerable or non-related files from other vendors (in green).

Those findings are reported as `CVE-2013-5637` and `CVE-2013-5638`. We were also able to confirm this vulnerability and extend the list of affected versions for one of these vendors.

Such manual vulnerability confirmation does not scale. Hence, in the future we plan to integrate static analysis tools for web applications [30, 11, 53, 38, 1] in our process.

6 Ethical Discussion

Large-scale scans to test for the presence of vulnerabilities often raise serious ethical concerns. Even simple Internet-wide network scans may trigger alerts from intrusion detection systems (IDS) and may be perceived as an attack by the scanned networks.

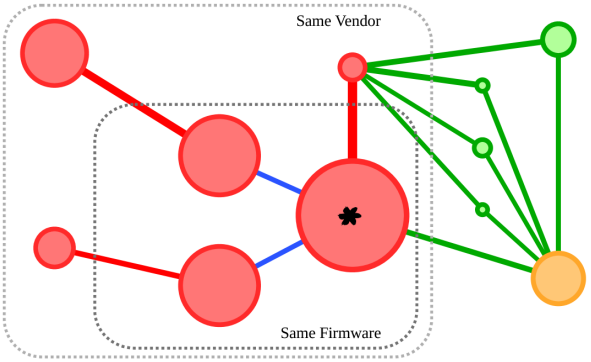


Figure 5: Fuzzy hash clustering and vulnerability propagation. A vulnerability was propagated from a *seed file (*)* to other two files from the same firmware and three files from the same vendor (in red) as well as one file from another vendor (in orange). Also four non-vulnerable files (in green) have a strong correlation with vulnerable files. Edge thickness displays the strength of correlation between files.

In our study we were particularly careful to work within legal and ethical boundaries. First, we obtain firmware images either through user submission or through legitimate distribution mechanisms. In this case, our web crawler was designed to obey the `robots.txt` directives. Second, when we found new vulnerabilities we worked together with vendors and CERTs to confirm the devices vulnerabilities and to perform responsible disclosure. Finally, the license of some firmware images may not allow redistribution. Therefore, the public web submission interface limits the ability to access firmware contents only to the users who uploaded the corresponding firmware image. Other users can only access anonymized reports. We are currently investigating ways to make the full dataset available for research purposes to well identified research institutions.

7 Related Work

Several studies have been proposed to asses the security of embedded devices by scanning the Internet. For instance, Cui et al. [28, 29] present a wide-scale Internet scan to first recognize devices that are known to be shipped with default password, and then to confirm that these devices are indeed still vulnerable by attempting to login into them. Heninger et al. [46] performed the largest ever network survey of TLS and SSH servers, showing that vulnerable keys are surprisingly widespread and that the vast majority appear to belong to headless or embedded devices. ZMap [37] is an efficient and fast network scanner, that allows to scan the complete Internet IPv4 address space in less than one hour. While the scans are not especially targeted to embedded devices, in our work we reuse the SSL certificates

scans performed by ZMap [36]. Similar scans were targeting specific vulnerabilities often present in embedded devices [40, 4]. Such wide-scale scans are mainly targeted at discovering online devices affected by already known vulnerabilities, but in some cases they can help to discover new flaws. However, many categories of flaws cannot be discovered by such scans. Some online services like Shodan [5] provide a global updated view on publicly available devices and web services. This easy-to-use research tool allows security researchers to identify systems worldwide that are potentially exposed or exploitable.

Unpacking firmware images is a known problem, and several tools for this purpose exist. Binwalk [41] is a firmware analysis toolbox that provides various methods and tools for extraction, inspection and reverse engineering of firmware images or other binary blobs. FRAK [26] is a framework to unpack, analyze, and repack firmware images of embedded devices. `FRAK` was never publicly released and reportedly supports only a few firmware formats (e.g., Cisco IP phones and IOS, HP laser printers). The Binary Analysis Toolkit (BAT) [45, 66] was originally designed to detect GPL license violations, mainly by comparing strings in a firmware image to strings present in open source software distributions. For this purpose `BAT` has to unpack firmware images. Unfortunately, as we show in Section 3, none of these tools are accurate and complete enough to be used *as is* in our framework.

There are many examples of security analysis of embedded systems [71]. Several network card firmware images have been analyzed and modified to insert a backdoor [33, 34] or to extend their functionality [16]. Davidson et al. [31] propose `FIE`, built on top of the `KLEE` symbolic execution engine, to incorporate new symbolic execution techniques. It can be used to verify security properties of some simple firmware images often found in practice. Zaddach et al. [70] describe Avatar, a dynamic analysis platform for firmware security testing. In Avatar, the instructions are executed in an emulator, while the IO accesses to the embedded system's peripherals are forwarded to the real device. This allows a security engineer to apply a wide range of advanced dynamic analysis techniques like tracing, tainting and symbolic execution.

A large set of firmware images of Xerox devices were reverse-engineered by Costin [24] leading to the discovery of hidden PostScript commands. Such commands allow an attacker to e.g., dump a device's memory, recover passwords, passively scan the network and more generically interact with devices' OS layers. Such attacks could be delivered to printers via web pages, applets, MS Word and other standard printed documents [23].

Bojinov et al. [18] conducted an assessment of the security of current embedded management interfaces. The study, conducted on real physical devices, found vulnerabilities in 21 devices from 16 different brands, including network switches, cameras, photo frames, and lights-out management modules. Along with these, a new class of vulnerabilities was discovered, namely *cross-channel scripting (XCS)* [17]. While XCS vulnerabilities are not particular to embedded devices, embedded devices are probably the most affected population. In a similar study, the authors manually analyzed ten Small Office/Home Office (SOHO) routers [48] and discovered at least two vulnerabilities per device.

Looking at insecure (remote) firmware updates, researchers reported the possibility to arbitrarily inject malware into the firmware of a printer [24, 27]. Chen [22] and Miller [58] presented techniques and implications of exploiting Apple firmware updates. In a similar direction, Basnight et al. [12] examined the vulnerability of PLCs to intentional firmware modifications. A general firmware analysis methodology is presented, and an experiment demonstrates how legitimate firmware can be updated on an Allen-Bradley ControlLogix L61 PLC. Zaddach et al. [72] explore the consequences of a backdoor injection into the firmware of a hard disk drive and uses it to exfiltrate data.

French and Casey [13] present fuzzy hashing techniques in applied malware analysis. Authors used `ssdeep` on *CERT Artifact Catalog* database containing 10.7*M* files. The study underlines the two fundamental challenges to operational usage of fuzzy hashing at scale: timeliness of results, and usefulness of results. To reduce the quadratic complexity of the comparisons, they propose assigning files into "bins" based on the block and file sizes. This approach, for their particular dataset and bins partitioning strategy, allowed for a given fuzzy hash to reduce the search space on average by 83.1%.

Finally, Bailey et al. [10] and Bayer et al. [14] propose efficient clustering approaches to identify and group malware samples at large scale. Authors perform dynamic analysis to obtain the execution traces of malware programs or obtain a description of malware behavior in terms of system state changes. These are then generalized into behavioral profiles which serve as input to an efficient clustering algorithm that allows authors to handle sample sets that are an order of magnitude larger than previous approaches. Unfortunately, this approach cannot be applied in our framework since dynamic analysis is unfeasible due to the heterogeneity of architectures used in firmware images.

8 Conclusion

In this paper we conducted a large-scale static analysis of embedded firmwares. We showed that a broader view on firmware is not only beneficial, but actually necessary

for discovering and analyzing vulnerabilities of embedded devices. Our study helps researchers and security analysts to put the security of particular devices in context, and allows them to see how known vulnerabilities that occur in one firmware reappear in the firmware of other manufacturers.

We plan to continue collecting new data and extend our analysis to all the firmware images we downloaded so far. Moreover, we want to extend our system with more sophisticated static analysis techniques that allow a more in-depth study of each firmware image. This approach shows a lot of potential and besides the few previously mentioned case studies it can lead to new interesting results such as the ones recently found by Costin et al. [25].

The summarized datasets are available at `http://firmware.re/usenixsec14`.

Acknowledgments

We thank the anonymous reviewers for their many suggestions for improving this paper. In particular we thank our shepherd, Cynthia Sturton, for her valuable time and inputs guiding this paper for publication. We also thank Pietro Michiardi and Daniele Venzano for providing access and support to their cloud infrastructure, and John Matherly of Shodan search engine for providing direct access to Shodan's data and resources.

The research leading to these results was partially funded by the European Union Seventh Framework Programme (contract Nr 257007 and project FP7-SEC-285477-CRISALIS).

References

[1] Audit PHP Configuration Security Toolkit.

[2] Define of backdoor string in DLink DI-524 UP GPL source code. `https://gist.github.com/ccpz/6960941`.

[3] Google Custom Search Engine API.

[4] Internet Census 2012 – Port scanning /0 using insecure embedded devices. `http://internetcensus2012.bitbucket.org`.

[5] SHODAN – Computer Search Engine. `http://www.shodanhq.com`.

[6] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[7] Slashdot: Backdoor found in TP-Link routers, March 2013.

[8] Download statistics for the wemo android application, February 2014. `http://xyo.net/android-app/wemo-JJUZgf8/`.

[9] Download statistics for the wemo iOS application, February 2014. `http://xyo.net/iphone-app/wemo-J1QNimE/`.

[10] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Z. Basnight, J. Butts, J. L. Jr., and T. Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76 – 84, 2013.

[13] L. Bass, N. Brown, G. M. Cahill, W. Casey, S. Chaki, C. Cohen, D. de Niz, D. French, A. Gurfinkel, R. Kazman, et al. Results of CMU SEI Line-Funded Exploratory New Starts Projects. 2012.

[14] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Symposium on Network and Distributed System Security*, NDSS '09. The Internet Society, 2009.

[15] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A View on Current Malware Behaviors. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[16] A. Blanco and M. Eissler. One firmware to monitor'em all. *Ekoparty*, 2012.

[17] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: Cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 420–431, New York, NY, USA, 2009. ACM.

[18] H. Bojinov, E. Bursztein, E. Lovett, and D. Boneh. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA*, 2009.

[19] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011.

[20] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware

Analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[21] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[22] K. Chen. Reversing and exploiting an Apple firmware update. *BlackHat USA*, 2009.

[23] A. Costin. Hacking Printers for Fun and Profit.

[24] A. Costin. PostScript(um): You've Been Hacked.

[25] A. Costin and A. Francillon. Short Paper: A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, WiSec '14. ACM, 2014.

[26] A. Cui. Embedded Device Firmware Vulnerability Hunting with FRAK. *DefCon 20*, 2012.

[27] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Proceedings of the 20th Symposium on Network and Distributed System Security*, NDSS '13. The Internet Society, 2013.

[28] A. Cui, Y. Song, P. V. Prabhu, and S. J. Stolfo. Brave New World: Pervasive Insecurity of Embedded Network Devices. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 378–380, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] A. Cui and S. J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-area Scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 97–106, New York, NY, USA, 2010. ACM.

[30] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, NDSS '14. The Internet Society, 2014.

[31] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 463–478, Berkeley, CA, USA, 2013. USENIX Association.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[33] G. Delugré. Closer to metal: reverse-engineering the Broadcom NetExtreme's firmware. *Hack.lu*, 2010.

[34] L. Duflot, Y.-A. Perez, and B. Morin. What if You Can'T Trust Your Network Card? In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 378–397, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] K. Dunham. A fuzzy future in malware research. *The ISSA Journal*, 11(8):17–18, 2013.

[36] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 291–304, New York, NY, USA, 2013. ACM.

[37] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 605–620, Berkeley, CA, USA, 2013. USENIX Association.

[38] B. Eshete, A. Villafiorita, and K. Weldemariam. Early Detection of Security Misconfiguration Vulnerabilities in Web Applications. In *Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security*, ARES '11, pages 169–174, Washington, DC, USA, 2011. IEEE Computer Society.

[39] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward Secure Embedded Web Interfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[40] HDMoore. Security Flaws in Universal Plug and Play: Unplug, Don't Play, 2013.

[41] C. Heffner. binwalk – firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images.

[42] C. Heffner. littleblackbox – Database of private SSL/SSH keys for embedded devices.

[43] C. Heffner. Breaking SSL on Embedded Devices, December 2010.

[44] C. Heffner. Reverse Engineering a D-Link Backdoor, October 2013.

[45] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding Software License Violations Through Binary Code Clone Detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 63–72, New York, NY, USA, 2011. ACM.

[46] N. Heninger, Z. Durumeric, E. Wustrow, and J. A.

Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[47] J. Hirsch and K. Bensinger. Toyota settles acceleration lawsuit after $3-million verdict. Los Angeles Times, October 25, 2013.

[48] Independent Security Evaluators. SOHO Network Equipment (Technical Report), 2013.

[49] IOActive. Critical DASDEC Digital Alert Systems (DAS) Vulnerabilities, June 2013.

[50] IOActive. stringfighter – Identify Backdoors in Firmware By Using Automatic String Analysis, May 2013.

[51] IOActive. Critical Belkin WeMo Home Automation Vulnerabilities, February 2014.

[52] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[53] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[54] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digit. Investig.*, 3:91–97, 2006.

[55] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[56] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008.

[57] P. C. Messina, R. D. Williams, and G. C. Fox. *Parallel computing works !* Parallel processing scientific computing. Morgan Kaufmann, San Francisco, CA, 1994.

[58] C. Miller. Battery firmware hacking. *BlackHat USA*, 2011.

[59] Nvidia. CUDA – Compute Unified Device Architecture Programming Guide. 2007.

[60] OpenwallProject. John the Ripper password cracker. `http://www.openwall.com/john/`.

[61] OWASP. Top 10 Vulnerabilities, 2013.

[62] V. Roussev. Data Fingerprinting with Similarity Digests. In *IFIP Int. Conf. Digital Forensics*, pages 207–226, 2010.

[63] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 851–862, New York, NY, USA, 2013. ACM.

[64] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 23–40, Berlin, Heidelberg, 2012. Springer-Verlag.

[65] J. V. Stough. distributed-python-for-scripting – DistributedPython for Easy Parallel Scripting.

[66] Tjaldur Software Governance Solutions. Binary Analysis Tool (BAT).

[67] A. Tridgell. rsync – utility that provides fast incremental file transfer.

[68] D. A. Wheeler. SLOCCount – a set of tools for counting physical Source Lines of Code (SLOC). `http://www.dwheeler.com/sloccount/`.

[69] xobs and bunnie. The Exploration and Exploitation of an SD Memory Card. *CCC – 30C3*, 2013.

[70] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, NDSS '14. The Internet Society, 2014.

[71] J. Zaddach and A. Costin. Embedded Devices Security and Firmware Reverse Engineering. *BlackHat USA*, 2013.

[72] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and Implications of a Stealth Hard-drive Backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 279–288, New York, NY, USA, 2013. ACM.

Exit from Hell? Reducing the Impact of Amplification DDoS Attacks

Marc Kührer, Thomas Hupperich, Christian Rossow, Thorsten Holz
Horst Görtz Institute for IT-Security, Ruhr-University Bochum

Abstract

Amplification vulnerabilities in many UDP-based network protocols have been abused by miscreants to launch Distributed Denial-of-Service (DDoS) attacks that exceed hundreds of Gbps in traffic volume. However, up to now little is known about the nature of the amplification sources and about countermeasures one can take to remediate these vulnerable systems. Is there any hope in mitigating the amplification problem?

In this paper, we aim to answer this question and tackle the problem from four different angles. In a first step, we monitored and classified amplification sources, showing that amplifiers have a high diversity in terms of operating systems and architectures. Based on these results, we then collaborated with the security community in a large-scale campaign to reduce the number of vulnerable NTP servers by more than 92%. To assess possible next steps of attackers, we evaluate amplification vulnerabilities in the TCP handshake and show that attackers can abuse millions of hosts to achieve 20x amplification. Lastly, we analyze the root cause for amplification attacks: networks that allow IP address spoofing. We deploy a method to identify spoofing-enabled networks *from remote* and reveal up to 2,692 Autonomous Systems that lack egress filtering.

1 Introduction

Distributed Denial-of-Service (DDoS) attacks have been known since many years [8,9,24,34] and they still constitute an important problem today. For a long time, DDoS attacks were hard to tackle due to their *semantic* nature: it is difficult to distinguish an actual attack from a sudden rise in popularity for a given service due to a flash crowd ("Slashdot effect"). A large body of literature is available on this topic and many DDoS detection mechanisms and countermeasures have been proposed over the years (e.g., [14,15,43]). Furthermore, advances in *Cloud computing* and load balancing techniques helped to mitigate this problem [17,18], and nowadays simple types of DDoS attacks such as *SYN* and *UDP flooding* are well-understood.

However, the adversaries evolved and modern DDoS attacks typically employ so called *amplification attacks*, in which attackers abuse UDP-based network protocols to launch DDoS attacks that exceed hundreds of Gbps in traffic volume [21, 22]. This is achieved via *reflective* DDoS attacks [31] where an attacker does not directly send traffic to the victim, but sends spoofed network packets to a large number of systems that reflect the traffic to the victim (so called *reflectors*). Often, attackers choose reflectors that send back responses that are significantly larger than the requests, leading to an increased (*amplified*) attack volume. We call such reflectors *amplifiers*. Recently, many types of such amplification attacks were discovered [33]. However, little is known about the nature of the amplifiers and about countermeasures one can take to remediate vulnerable systems.

In this paper, we address this problem and study the root causes behind amplification DDoS attacks. We tackle the problem from four different angles and provide empirical measurement results based on Internet-scale scanning to quantify the problem.

In a first step, we want to understand the nature of amplifiers and determine which kinds of systems are vulnerable. Previous work on empirically understanding DDoS attacks typically focused on ways to estimate the size of the problem and understanding the infrastructure behind such attacks [1, 5, 26]. To increase the understanding of amplification attacks, we utilized protocol-specific fingerprinting to reveal as much information as possible from systems that can be abused on the Internet. More specifically, we enumerated the amplifier sources for seven network protocols and performed large-scale scans to collect information about vulnerable systems. This enables us to categorize the types of devices that can be abused in the wild. We found that there is a large diversity of vulnerable devices and analyzed their properties. For example, we found 40.8% of the vulnerable NTP hosts to run Cisco IOS, an OS that is deployed on Cisco network devices.

Based on these insights on amplifiers, a viable next step is to reduce the number of vulnerable systems on the Internet. Previous work on that topic mainly focused

on understanding botnet Command & Control servers that are used to orchestrate classical DDoS attacks [5]. However, modern amplification attacks use a completely different modus operandi. We contributed to a global security notification procedure where our scanning results were used to notify NOCs and CERTs of hundreds of large ISPs worldwide about NTP servers vulnerable to amplification attacks. Furthermore, we collaborated with security organizations in order to create advisories that describe the technical background and approaches to solve the problem. We analyzed the remediation success of these measures and found that the number of NTP servers vulnerable to `monlist` amplification dropped by 92% in a 13-week period between November 2013 and February 2014. We closely analyzed this effect and found that especially vulnerable NTP servers within ARIN have been mitigated, while other geographic regions lag behind.

Since it seems feasible to significantly reduce the number of amplifiers, a third angle of the problem is an analysis of potential attack vectors that adversaries could abuse in the future. We start with the basic insight that up to now UDP-based protocols are leveraged by attackers, since these protocols provide large amplification factors. We study a completely different kind of amplification attacks, namely TCP-based ones. Surprisingly, even TCP can be abused for amplification attacks, despite the fact that this protocol uses a 3-way handshake. This is due to the fact that certain TCP stacks retransmit `SYN/ACK` packets multiple times (some 20x or more) when they presume that the initial `SYN/ACK` segment was lost. Thus an amplification of 20x or more is possible. Empirical scan results suggest that there are hundreds of thousands of systems on the Internet that can be abused this way. We performed protocol-specific fingerprinting to learn more about the nature of such devices.

As a fourth angle of the problem, we analyzed the root cause behind amplification attacks: if a given network does not perform *egress filtering* (i.e., verifies that the source IP address in all outbound packets is within the range of allocated internal address blocks, see BCP 38 [13] for details), an attacker can spoof packets and thus initiate the first step of reflective DDoS attacks. Identifying such networks is a challenging problem [12] and existing solutions rely on a client deployed in the network under test [3, 36]. We utilize a novel remote test based on DNS proxies that enables us to identify thousands of Autonomous Systems that support IP spoofing.
To summarize, our contributions are as follows:

- We performed Internet-wide scans to identify and monitor all relevant potential amplifiers for seven network protocols vulnerable to amplification attacks. We fingerprint and categorize these systems, showing a high diversity in the amplifier landscape.
- We study the success of a global security notification campaign to alert administrators of vulnerable NTP servers and show the benefits and limitations of such large-scale initiatives.
- Aiming to assess further amplification DDoS techniques, we identify TCP as an alternative source for amplification—despite its three-way-handshake protocol. We reveal millions of systems that can be abused to amplify TCP traffic by a factor up to 20x.
- Finally, we aim to tackle the root cause for amplification DDoS attacks: networks that do not perform egress filtering and thus allow IP address spoofing. We deploy a *remote* scanning technique and find up to 2,692 ASes that permit spoofed IP traffic.

Paper Outline. The paper is organized as follows. In Section 2, we define the threat model and outline our scanning setup to perform Internet-wide scans. We then shed light onto the landscape of hosts that are vulnerable to UDP-based amplification DDoS attacks. In Section 3, we detail the effects of our NTP case study. Section 4 tackles the problem of TCP-based amplifiers, demonstrating that the TCP three-way-handshake can be abused for amplification attacks. In Section 5, we introduce a novel mechanism to identify networks that allow IP address spoofing. Section 6 reviews prior work and we conclude this paper in Section 7.

2 Amplification DDoS

We begin with an analysis of the threat landscape. To this end, we first review the general threat model before we analyze different aspects of amplification DDoS attacks. More specifically, we study the amplifier magnitude, measure what kinds of devices can be abused on the Internet, and determine the churn of amplifiers.

2.1 Threat Model

The scope of this work are amplification DDoS attacks. In such an attack, a miscreant abuses public systems (such as open recursive DNS resolvers) to reflect attack traffic to a DDoS victim [31]. In particular, she abuses hosts that not only reflect but also amplify the traffic. Typically, the attacker chooses connection-less protocols in which she can send relatively small requests that result in significantly larger responses. By spoofing the source of the traffic (i.e., impersonating the victim), she can enforce that the public systems—unwillingly—amplify and reflect traffic to the victim. Prior work has revealed that at least 14 UDP-based protocols are vulnerable to such abuse [33]. These protocols offer severe amplification rates—in the worst case, as with the `monlist` feature in NTP, they amplify traffic by a factor of up to 4,670.

2.2 Amplifier Magnitude

In this paper, we try to shed light onto the landscape of *amplifiers*, i.e., hosts that are vulnerable to amplification abuse. As a first step, we enumerate and observe these amplifiers in the IPv4 address space. That is, we performed Internet-wide scans for a subset of the vulnerable protocols: DNS, SNMP, SSDP, CharGen, QOTD, NTP, and NetBIOS. We chose to monitor these protocols, as prior work only approximated the amplifier landscape for them. The amplification vulnerabilities of these seven protocols can be abused by attackers to launch severe amplification attacks. In addition, all these seven protocols run server-side, thus hosts running such protocols are seemingly better connected and more stable in terms of IP address churn than hosts of end users.

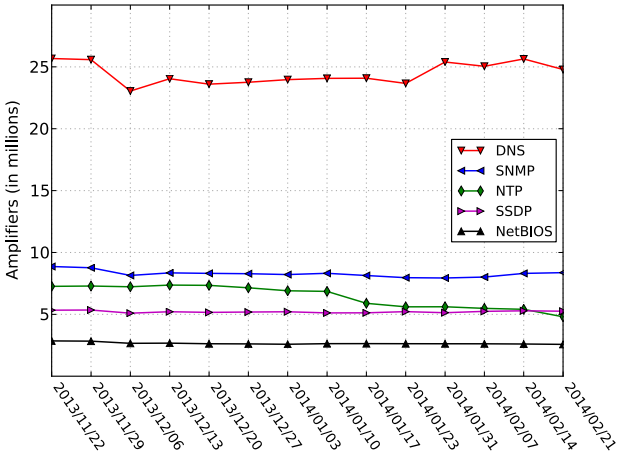


Figure 1: Trend of UDP-based amplifiers

Scanning Setup. We developed an efficient scanner to identify amplifiers for these protocols in Internet-wide scans. In order to respect good scanning practices as suggested by Durumeric et al. [11], we limit the number of requests that a particular network receives. For this reason, we compute the scan targets as a pseudo-random permutation of the entire IPv4 address space (except the IP address 0.0.0.0). That is, we use a linear feedback shift register (LFSR) to compute the order of the $2^{32}-1$ IPv4 addresses to be scanned. In order to avoid to become blacklisted, we refrained from aggressive scanning and distributed the scans over 48 hours. In addition, we set up a reverse DNS (rDNS) record for our scanner and configured a web server that presents project information and an explanation how to opt-out from our scans.

For each of the protocols, we send a request that can be used to amplify traffic. That is, we send NTP `version` requests, SSDP `SEARCH` requests, SNMP v2 `GetBulk` requests, DNS `A` lookups, and NetBIOS' default name lookup. We ran the scans on a weekly basis from Nov 22, 2013 to Feb 21, 2014 to observe potential changes in terms of amplifiers. We chose to use the weekends for our scans so that the load of both our scanning network and the scanned networks have less impact on business activities. In the case of CharGen and QOTD, we refrained from repeating the scans, as the number of amplifiers was too low to justify repeated full scans.

During the course of our scans, we received 90 emails from administrators asking about the scanning experiments. Adhering to these requests, we excluded 91 IP prefixes and 30 individual IP addresses (about 3.7 million IP addresses in total) after administrators asked us to do so. To allow comparisons between two scans, we ignored these IP addresses in all of our scans, i.e., even if they were not blacklisted at the beginning.

Table 1: Intersection of potential amplifiers based on the Internet-wide scan on Nov 22, 2013

	Intersection (in %)						
Protocol	*DNS*	*CharGen*	*NetBIOS*	*NTP*	*QOTD*	*SNMP*	*SSDP*
DNS	-	0.0	0.5	0.2	0.0	11.6	2.1
CharGen	1.7	-	2.4	20.0	4.0	9.4	1.3
NetBIOS	4.7	0.1	-	0.6	0.2	1.8	5.9
NTP	0.9	0.3	0.2	-	0.0	3.2	0.1
QOTD	14.0	11.8	18.5	8.4	-	4.2	8.5
SNMP	33.5	0.1	0.6	2.7	0.0	-	0.2
SSDP	9.9	0.0	3.1	0.2	0.1	0.4	-

Results. Figure 1 illustrates the number of identified amplifiers per protocol. By far the highest number of amplifiers was found for open recursive DNS resolvers, slightly fluctuating between 23 and 25.5 million systems. For most of the other protocols, the number of amplifiers is quite constant. An exception are NTP amplifiers, whose popularity constantly decreases, a phenomenon that we describe in detail in Section 3. None of the protocols (except for the two legacy protocols CharGen with 107,725 and QOTD with 36,609 vulnerable hosts) had fewer than 2.5 million amplifiers, showing a large landscape of hosts that can be abused.

Quite interestingly, some systems run multiple vulnerable services. Table 1 shows the intersection between the individual protocols relative to the overall number of amplifiers for the protocols specified in the first table column. The largest overlap is between DNS and SNMP: a third of the public SNMP hosts also run open recursive DNS resolvers. Note that the table is not symmetric, which, for example, reveals that less than 11.6% of the open DNS resolvers also run unprotected SNMP daemons. For most of the other protocols the intersection is negligible, though. This means that the number of amplifiers basically sums up. We measured almost 46 million amplifiers for all scanned UDP-based protocols.

Table 2: Results of the device fingerprinting for the amplifiers identified on Nov 22, 2013

	Hardware (in %)				Architecture (in %)					Operating System (in %)										
Protocol	*Router*	*Embedded*	*Others*	*Unknown*	*x86*	*MIPS*	*PowerPC*	*Others*	*Unknown*	*Unix*	*Linux*	*Ubuntu*	*FreeBSD*	*Windows*	*ZyNOS*	*Cisco IOS*	*Junos*	*NetOS*	*Others*	*Unknown*
DNS	9.7	5.7	0.6	84.0	0.6	7.0	0.0	0.4	92.0	3.6	3.4	0.0	0.0	0.8	7.5	0.1	0.0	0.0	1.1	83.5
NetBIOS	0.7	1.3	2.0	96.0	87.6	0.1	0.0	0.0	12.3	0.4	0.1	0.0	0.0	87.3	0.3	0.0	0.0	0.0	0.7	11.2
NTP	44.8	0.5	2.4	52.3	9.6	18.4	6.9	1.1	64.0	18.2	26.8	0.0	4.7	0.2	0.0	40.8	2.9	0.0	1.7	4.7
SNMP	66.5	10.4	3.1	20.0	2.9	44.9	1.1	3.1	48.0	1.5	11.4	0.1	0.1	0.8	17.8	2.2	0.0	0.0	8.7	57.4
SSDP	94.3	2.9	2.2	0.6	1.5	2.7	0.0	0.1	95.7	1.8	36.0	5.5	0.0	1.3	0.7	0.0	0.0	19.3	1.8	33.6

2.3 Amplifier Classification

Observing the magnitude of the problem, we wondered what kinds of systems allow for such amplification vectors. In an attempt to answer this question, we use protocol-specific fingerprinting to reveal as much information as possible from these systems. That is, we generate device fingerprints by inspecting the replies from the amplifiers during our UDP scans. We dissect the responses of each host and protocol individually to classify systems in three categories: the underlying hardware (e.g., routers, desktop computers, or printers), the system architecture (such as x86, MIPS, or PowerPC), and the operating system.

Fingerprinting Setup. We manually compiled 1,873 regular expressions that allow a fine-granular generation of fingerprints. We further leverage Nmap service probes [27] to fingerprint the NetBIOS protocol. For NetBIOS, we also focus on the structure of the payload to obtain information about the OS. NTP `version` responses reveal the processor type, OS, and the version of the running NTP daemon. To generate fingerprints for the SNMP protocol, we analyze the object identifier values (OID) in the responses. For SSDP, the responses contain text fragments resembling HTTP headers that provide system information in the `Server` header field. Additionally, SSDP headers include Unique Service Name (USN) and Search Target (ST) fields, providing more general information about a device.

We improve the coverage of our UDP fingerprints by scanning the amplifiers for common TCP-based protocols. We use FTP, HTTP, HTTPS, SSH, and Telnet to leverage information in protocol banners and text fragments. We synchronize the TCP and UDP scans, hence once a full Internet-wide UDP scan is finished, we initiate a follow-up TCP scan for hosts that are found to be an amplifier for at least one UDP protocol.

Results. Table 2 depicts the fingerprint results obtained for the amplifiers found on Nov 22, 2013. For reasons of brevity, we summarize fingerprint details with less than 2% share to *Others*. Note that the category "router" also includes gateways, switches, and modems as many of these devices provide similar features.

The best results for the OS classification are achieved for NTP. We find that 40.8% of the vulnerable NTP hosts run Cisco IOS, an OS that is deployed on Cisco devices such as business routers and switches. We further identify 1,267,008 amplifiers (17.4%) running Linux on MIPS and 357,076 devices (4.9%) running Linux on PowerPC. These two combinations are common for consumer devices such as routers and modems. The majority of NTP amplifiers thus run on networking equipment.

Similarly, two thirds of the SNMP amplifiers are routers. With a share of 17.8%, the ZyNOS system stands out—apparently running unprotected SNMP services per default. But we also observe a wide distribution of other SNMP devices. This includes 58,000 office printers (0.7%), 51,037 firewall appliances (0.6%), and 40,061 network cameras (0.5%). Routers are even more prominent among SSDP hosts with a share of about 94.3%. This shows that at least three of the analyzed protocols are overly prominent on routers.

On the contrary, the vast majority of NetBIOS amplifiers run Windows on x86, a typical setup of desktop computers. Since the Conficker outbreak in 2008, it is known that millions of Windows systems on the Internet are reachable via the NetBIOS [32] protocol.

Unfortunately, DNS provides only limited fingerprint information and we thus had to solely rely on the TCP fingerprints to classify DNS servers. However, most DNS servers did not run TCP services, resulting in a high number of uncategorized hosts. Even if TCP services were accessible, the provided information was often too generic (e.g., banners as "Apache", "SSH-2.0-OpenSSH", or "FTP Server"). However, we could identify 5.4% of the hosts (1,388,348) as MIPS-based routers with ZyNOS, which is common for broadband routers distributed by manufacturer ZyXEL.

A high diversity of amplifiers is attested when looking at smaller clusters. For example, we find 695 vulnerable devices to be running *Miele Logic*, a payment system for Miele devices such as washing machines. Similarly, we identify 9,224 amplifiers running server man-

Table 3: Amplifier churn rate per protocol

	Initial Scan	Week 1		Week 13	
Protocol	(#)	(#)	(%)	(#)	(%)
DNS	25,681,450	12,190,302	47.5	8,263,508	32.2
NetBios	2,853,213	1,455,351	51.0	979,266	34.3
NTP	7,269,015	6,859,043	94.4	4,222,060	58.1
SNMP	8,866,748	4,939,118	55.7	3,411,563	38.5
SSDP	5,336,107	3,088,148	57.9	2,067,830	38.8

agement systems (like iLO, iDRAC, or IPMI). We further find 51,351 Digital Video Recorders, 7,739 Power Distribution Units, and 20,927 Network Attached Storage devices (NAS) to be vulnerable to amplification.

Ambiguous Fingerprints. We had to resolve a few conflicts when combining the fingerprints from multiple protocols. For NTP amplifiers, we find valid TCP fingerprints for 1,919,932 hosts, while conflicts emerge for 9,945 IP addresses (0.5%). For SNMP, we leverage TCP data for 2,042,541 amplifiers while obtaining 31,346 conflicts (1.5%). We presume that these conflicts were caused by responses from "border" devices such as routers that host some services themselves (e.g., SNMP and SSH), while requests for FTP or HTTP were forwarded to the devices connected to the router, resulting in multiple fingerprints for a single IP address. To resolve these conflicts, we assign a lower priority to TCP fingerprints when classifying the amplifiers. In addition, we refrained from aggregating the individual UDP fingerprints to one large set, as the overlap between the UDP protocols is low anyway (cf. Table 1).

2.4 Amplifier Churn

An important aspect from the attacker's point of view is how fast the set of amplifiers changes. An up-to-date list of reliable amplifiers is key to achieving a high impact during an attack. For this reason, we measure the *churn rate* of the amplifiers per protocol, which shows how fast a list of amplifiers becomes outdated. That is, we enumerate the amplifiers based on their IP addresses on Nov 22, 2013 and check if these hosts are still vulnerable for amplification attacks in the subsequent weeks.

Table 3 lists the numbers of amplifiers for the five UDP protocols that we monitored on long term. Figure 2 illustrates the ratio of amplifiers that are still reachable at the same IP address over time. For most protocols (DNS, NetBIOS, SNMP, and SSDP) the churn of amplifiers is quite high: only about 50% of the initial hosts are still reachable after one week. After the second week, we again observe a minor decrease, resulting in a total of 40 - 50% of available amplifiers for each protocol. For the following weeks we find the number of amplifiers to reach an almost steady level.

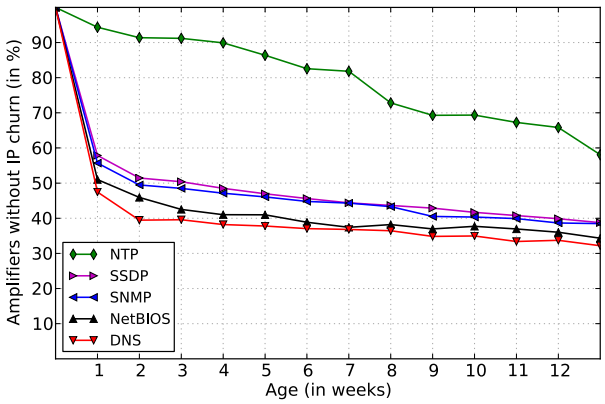


Figure 2: IP churn of potential amplifiers

To understand the nature of the significant drop after the first week and obtain knowledge about the vulnerable systems still reachable after 13 weeks, we leverage our fingerprinting techniques. We find the amplifiers that became outdated within the first week to be mostly connected via consumer routers (e.g., 78.8% for SNMP). We assume that these routers are connected via DSL with low IP address lease times, causing the rapid breakdown rate after one week [37]. To confirm this assumption, we aggregated reverse DNS records for a random sample of 100,000 unreachable amplifiers and checked for common indicators of dial-up connections (i.e., the appearance of tokens such as "dialup", "dyn", or "pool"—we further manually verified the Top 5 providers not providing indicators in the rDNS data). We indeed found at least 82.8% of the IP addresses to be linked to connections with dynamic IP address assignment. This means that an attacker needs to frequently re-scan for amplifiers or otherwise risks to decrease the impact of her attacks.

The amplifiers that are still reachable after 13 weeks presumably have longer lease times or static Internet connectivity. For example, we can see a clear distinction between countries in which SSDP hosts disappeared after a week (e.g., China, Argentina, Russia) and countries in which most hosts are still reachable after 13 weeks (e.g., Korea, United States, Canada). While only 3.4% of the Chinese amplifiers were still reachable after 13 weeks, still more than 69% of the Canadian amplifiers were available. This shows that the geolocation of Internet links (and thus the risk to face IP address churn) highly influences the availability of amplifiers.

Interestingly, the NTP protocol draws a completely different picture. Given a fixed list of vulnerable hosts, the ratio of available NTP amplifiers decreases at a negligible rate. After four weeks, an attacker can still abuse approximately 90% of the initial NTP amplifiers. After 13 weeks, still 58.1% of the initially-scanned hosts are

reachable. Of the 41.9% decrease after 13 weeks, many systems presumable disappeared because of our NTP amplifier notification campaign (cf. Section 3)—and not because of IP churn. The actual churn is thus even lower and significantly differs from churn in the other protocols. In contrast to the vulnerable amplifiers of the other protocols, more than 40% of the NTP amplifiers are running Cisco IOS that is commonly distributed on business routers and switches with static IP addresses. We further find 53.7% of the vulnerable hosts to be located within the United States (31.3%), South Korea (13.0%), and Japan (9.4%) for which the typical IP lease times for broadband are above average.

3 Case Study: NTP Amplification

After inspecting the amplification attacks in general, this section focuses on NTP, which we consider by far the worst among all known vulnerable protocols. NTP is a promising amplification vector for an attacker for three reasons. First, NTP server implementations allow for amplification factors of up to 4,670 [33]. Attackers can abuse the `monlist` feature in popular `ntpd` versions, which requests a list of up to 600 NTP server clients in about 44kB UDP payload. Second, as we have seen in Section 2.4, NTP servers have minimal IP address churn. Lastly, NTP offers even further amplification vectors. For example, the NTP `version` request reveals a verbose system fingerprint (OS, architecture, server info) of the NTP server, allowing about 24-fold amplification.

Attackers have already practically demonstrated the impact of NTP attacks. For example, in February 2014, CloudFlare observed a 400 Gbps attack against a French hosting provider [22]—the largest DDoS attack observed so far. If the attacker had even more resources (in particular bandwidth) to send spoofed `monlist` requests, the impact of such an attack could have been even higher.

Luckily, NTP servers can be configured such that the `monlist` requests are disabled for unauthorized users, and more recent `ntpd` versions protect this feature with a proper session handshake. These changes typically do not bring disadvantages for the administrators, while they eliminate the amplification vector. Even disabling functionality like `monlist` does not break time synchronization. But although secure configurations are well-documented, most administrators are not aware of the amplification vulnerabilities and operate NTP servers in (sometimes bad) default configurations. From a security-perspective this raises several urging questions: once we found amplification vulnerabilities, how can we reduce the number of amplifiers? Can we notify administrators? How effective would such a notification procedure be?

3.1 NTP Amplifier Notifications

In a large-scale campaign, we have launched a global notification procedure to alert NTP administrators about the amplification problems. We thankfully cooperated with many parties striving towards the same goal: reducing the number of NTP amplifiers.

Datasets. We define two datasets of NTP amplifiers. NTP_{ver} contains all NTP servers that reply to `version` requests, i.e., systems that are "less" vulnerable to amplification abuse. This is the same dataset that we fingerprinted in Section 2. As a subset of this, NTP_{mon} contains the NTP servers that also support the `monlist` requests, i.e., systems that allow for more "severe" amplification.

Campaign. We collaborated with security organizations in order to create technical advisories that describe how to solve the amplification problems in NTP. This resulted in public advisories of CERT-CC [42] and MITRE [25]. Due to the high number of vulnerable Cisco devices for NTP amplification (cf. Table 2), we also contacted Cisco which resulted in a public advisory of Cisco's PSIRT [7]. The advisories describe how to disable the `monlist` feature in typical NTP server implementations (such as `ntpd`). The same configuration change also disables `version` responses. Thus, in principle, the advisories help to reduce the number of servers in both datasets, NTP_{ver} and NTP_{mon}.

In addition, we distributed lists of IP addresses of the systems in NTP_{mon} among trusted institutions. For example, we shared our data with direct contacts in NOCs and CERTs of hundreds of large ISPs worldwide. Furthermore, we cooperated with data clearing houses (e.g., TrustedIntroducer [41] and ShadowServer [35]) that informed their subscribers. Lastly, we informed the NTP Pool Project [28] about misconfigured hosts in the public pool of NTP servers and synchronized our notifications with the OpenNTPProject [30] to start the announcements simultaneously.

We did not actively notify systems that are only in NTP_{ver} for two reasons. First, we saw an urgent need to close the amplifiers in NTP_{mon}, as the `monlist` amplification is in the order of magnitudes higher than of other NTP features. Second, we can then compare the efficiency of advisories (which affect both datasets) with the effects of active and personalized notifications (which affect only NTP_{mon}).

3.2 Analyzing the Remediation Success

We ran weekly scans for NTP amplifiers to observe the developments over time. Figure 3 shows the number of NTP amplifiers per week and marks important events.

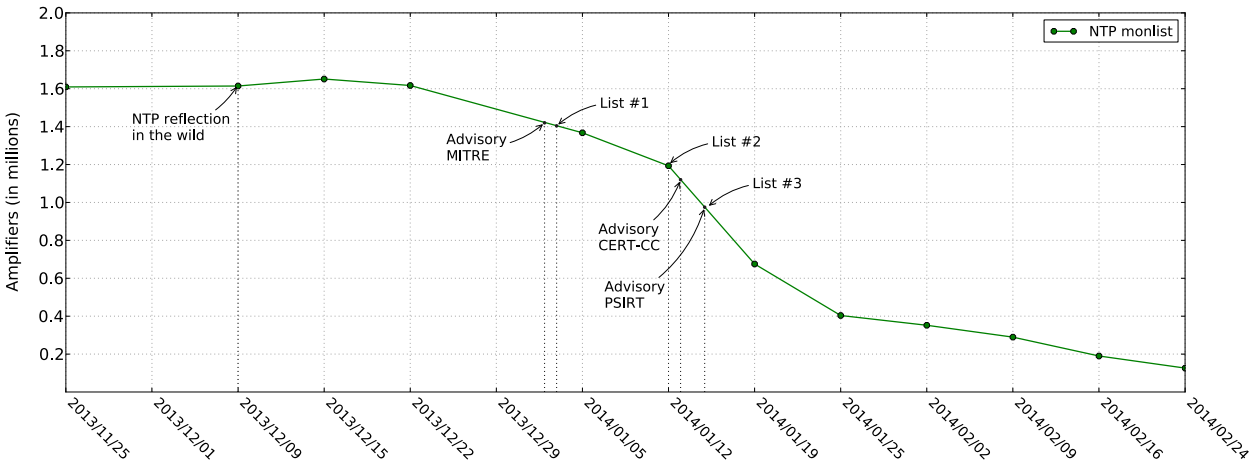


Figure 3: Trend of NTP_{mon} amplifiers

At their peaks on Dec 15, 2013, we tracked 7,364,792 servers in NTP_{ver} and 1,651,199 servers in NTP_{mon}, respectively. The number of amplifiers in NTP_{mon} is steady at first, also after Symantec has released a blog article about the attacks late December 2013 [39]. However, the number of amplifiers starts to drop (15.4%) right after we released the CVE and shared a first (incomplete) list of IP addresses. A second major drop (43.4%) happened during the week we distributed twice an updated (and complete) list of potential amplifiers and two other advisories were released. After publishing further weekly notifications (omitted from the graph for better readability) we have observed a steady decrease of amplifiers.

At the end of our measurements on Feb 24, 2014, the number of amplifiers reached a local minimum at 4,802,212 (NTP_{ver}) and 126,080 (NTP_{mon}). Compared to the peak numbers, this constitutes significant drops in both datasets. The number of amplifiers in NTP_{ver} decreased by 33.9%, a success likely stemming from the advisories and recent publicity on NTP attacks in general. However, looking at the development of *severe* amplifiers shows how successful global notification efforts can be: the number of amplifiers (NTP_{mon}) dropped by 92.4% with an ongoing decrease.

To verify whether the number of amplifiers for NTP_{mon} still decreases continuously, we performed another Internet-wide scan on Jun 20, 2014 and find 87,463 hosts still vulnerable to `monlist` amplification, i.e., a decrease of almost 40,000 hosts since Feb 24, 2014.

Fingerprinting. We compared the fingerprints of the NTP_{mon} datasets at the start and end of our measurements. We clearly observe decreasing numbers for all architectures, OSes, and hardware types. Interestingly, the ratio of MIPS-based amplifiers dropped from 47.2% to 19.1%, while the ratio of x86-based systems increased from 40.2% to 58.0%. Similarly, 23.0% of the devices of a popular router manufacturer remain vulnerable—a value standing out from the average decrease. On absolute scale, though, the numbers drop across all fingerprints, indicating that the clean-up was not driven only by a single device type or manufacturer.

Table 4: Decrease of NTP_{mon} amplifiers per country

	Amplifiers (in #)			Remaining
Country	Nov 22, 2013	Feb 24, 2014	Decrease	(in %)
US	1,073,666	28,415	1,045,251	2.6
KR	88,289	16,183	72,106	18.3
RU	58,519	11,476	47,043	19.6
DE	50,627	4,793	45,834	9.5
CA	36,070	1,881	34,189	5.2
CN	32,995	4,172	28,823	12.6
JP	29,915	2,777	27,138	9.3
GB	24,408	2,741	21,667	11.2
UA	19,270	2,716	16,554	14.1
BR	13,900	2,719	11,181	19.6
TW	13,362	6,397	6,965	47.9
NL	13,122	3,934	9,188	30.0
FR	12,992	4,557	8,435	35.1
CZ	11,825	1,226	10,599	10.4
PL	10,891	1,960	8,931	18.0

Geographic Distribution. We also investigated the geographical distribution of the amplifiers. For this, we used the MaxMind GeoIP database [23] to assign a country to the IP address of an amplifier. We then compared how the numbers of amplifiers evolve in single countries. Table 4 lists the remaining amplifiers of the 15 countries, which had the most amplifiers in Nov 2013. The clean-up was—on relative scale—most successful in the US, where the number of amplifiers decreased to only 2.6%. In other countries like Taiwan the number decreased only to 47.9%. These differences may be caused by the number and quality of direct contacts we had in the US compared to Taiwan: we admittedly had more contacts in the

Table 5: Decrease of NTP_{mon} amplifiers per RIR

	Amplifiers (in #)			Remaining
RIR	Nov 22, 2014	Feb 24, 2014	Decrease	(in %)
ARIN	1,112,422	30,766	1,081,656	2.8
RIPE	283,991	53,324	230,667	18.8
APNIC	202,719	38,122	164,597	18.8
LACNIC	21,721	5,075	16,646	23.4
AFRINIC	7,495	920	6,575	12.3

US and Europe compared to the rest of the world. But it also shows that the current network of CERTs is not perfectly connected to share our information equally in all countries. For example, also European countries like France (35.1%) and Austria (47.1%) lag behind the average decrease. However, on an absolute scale, the situation is different. While the number of NTP amplifiers was clearly reduced in the US, still 28,415 systems remain vulnerable to `monlist` amplification.

Table 5 shows the absolute numbers of NTP servers per Regional Internet Registry (RIR), which (very roughly) indicates the continent of the amplifiers. It shows that we face a global problem, but also proves that all regions in the world have acknowledged the problems.

Per-Provider Statistics. A closer look at the Autonomous Systems (AS) distribution sheds light onto how amplifiers have been closed. Of the 96 ASes that had at least 1,000 amplifiers, about half have shut down more than 95% of the amplifiers. This shows that many providers either enforce most amplifiers in their networks to be shut down or successfully filter NTP traffic at the network level. More specifically, we identified 73 ASes (0.44% of all ASes we observed during our monitoring period) that had more than 100 NTP servers in one week, and did not have a single vulnerable server left open in the subsequent weeks. This strongly suggests that these providers perform network-level filters, as it is unlikely that so many individual servers were all cleaned up within a few days. Nine ASes left open more than half of the amplifiers, i.e., these providers do very little to mitigate the threat. We are currently establishing individual contacts with the least-active ASes and hope to understand the reasons for the remaining amplifier landscape in their networks.

Result Verification. We verified if the drop in NTP amplifiers is not caused by networks blocking our scanner [10]. This can already be seen in the amplifier trend graph (Figure 1), in which the number of amplifiers for other protocols remains almost constant. To be sure, we scanned for NTP amplifiers from a secondary host in a different `/16` network. The primary scanner indeed missed 8.6% of the amplifiers that the secondary scanner found. We manually investigated this and found 904 networks (`/20`) that have at least five amplifiers that the primary scanner missed—indicating that some networks do blacklist our scanner. While our primary scanner has thus missed amplifiers, these systems make up an almost-negligible part of the 92.2% decrease of amplifiers.

3.3 Lessons Learned

Summarizing, the campaign to reduce the number of NTP `monlist` amplifiers was quite effective and showed remediation successes for almost 95% of the vulnerable hosts just after 6 months. As such, it would be interesting to see recipes to repeat this success in similar campaigns for other security-critical issues, such as amplification vulnerabilities in other protocols or even unrelated, but security-critical problems like the `heartbleed` vulnerability in OpenSSL. Figure 3 clearly shows that the counteractions (advisories and IP address lists) correlate with the decrease in numbers of amplifiers.

Unfortunately, it is impossible to proof causality, in particular, to see which IP address distribution channels or which advisories were most effective. However, in our conversations with providers we had the impression that it helps to repeatedly point out the problem. Further, it may not be sufficiently effective to have public advisories that nobody reads. Instead, we found that communication was key to motivate CERTs and providers to act accordingly. Once we reached out to CERTs and providers, it typically was no problem to close the vulnerable hosts.

On the negative side, though, we experienced that the Internet community is not well-prepared for such campaigns. Although we were quite well-connected with nationally and internationally operating CERTs and providers, it is hard to reach out to all providers individually. If providers and CERTs were better connected to non-profit data clearing houses (like `shadowserver.org`), vulnerability notifications could be sent out more efficiently.

4 TCP-based Amplification Attacks

In the previous section, we have shown that we can have an influence on the amplifier landscape. As such, we introduce next steps attackers may take once we "fix" all the protocols that have been documented to be vulnerable for amplification attacks [33]. Given the connection-less nature of UDP, it comes as no real surprise that UDP-based protocols may allow for amplification attacks.

In this section, we analyze to what extent TCP allows for amplification attacks similar to the UDP-based attacks. TCP is a connection-oriented protocol, in which early on (i.e., during the handshake) the IP addresses of both communication parties are implicitly verified via

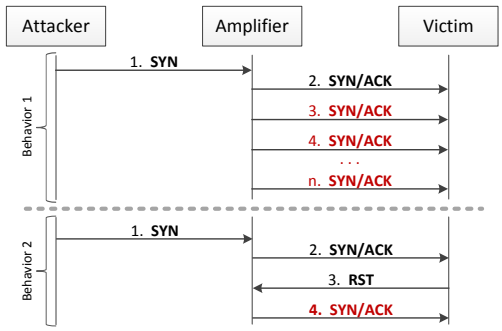


Figure 4: Amplification abuse in the TCP handshake

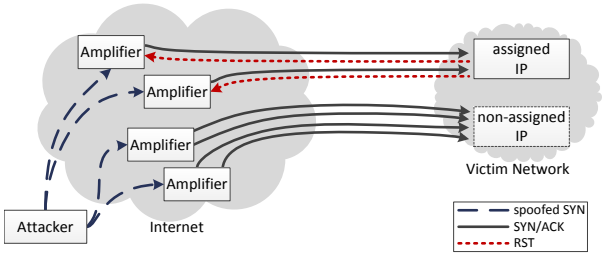


Figure 5: Attacker's choice of targets

initially-random TCP sequence numbers. We evaluate how TCP can be abused for amplification regardless of the TCP three-way-handshake.

From the viewpoint of the attackers, abusing TCP brings multiple benefits. First, providers cannot easily block or filter TCP traffic related to well-known protocols (such as HTTP), as compared to protocols that are less critical (such as CharGen, QOTD, or SSDP). In addition, it is hard to distinguish attacks from normal traffic in a stream of TCP control segments, while providers can deploy payload-based filters for attack traffic from many UDP-based protocols. Lastly, there are millions of potential TCP amplifiers out there and "fixing" them seems like an infeasible operation.

4.1 TCP Amplification Background

TCP initiates a connection with a three-way-handshake, which works as follows: a client willing to start a TCP connection sends a `SYN` segment to a server and this packet contains a random sequence number seq_A. If the server is willing to accept the client, it responds with a `SYN/ACK` segment, in which the acknowledgment number is set to $seq_A + 1$ and a random sequence number seq_B is added as well. In the third step, the client completes the connection setup by sending a final `ACK` to the server where the sequence number is set to $seq_A + 1$ and the acknowledgement number is set to $seq_B + 1$.

At first sight, TCP thus does not allow amplification: all segments are of the same size and no data bytes are exchanged before the handshake is finished. Assuming that the server draws TCP sequence numbers at random, there is no practical way to complete the handshake with IP-spoofed traffic. If the client address is spoofed, theoretically only one single `SYN/ACK` is sent to a potential victim. While this allows to reflect traffic, it does not amplify the traffic and therefore does not attack a victim with more bytes than sent by an attacker.

In practice, though, TCP connections encounter packet loss. TCP stacks thus deploy segment retransmissions, i.e., they retransmit segments that have not been acknowledged by the other party. We noticed that retransmissions also occur *during the handshake*. Many popular TCP stacks resend `SYN/ACK` segments until: (i) an `ACK` is received (and the connection is successfully established), (ii) a threshold is met (and the connection times out), or (iii) the connection is closed by the client (e.g., via a `RST` segment). In face of amplification attacks, this is problematic, as the client's IP address is not validated until the handshake is complete.

Figure 4 illustrates two typical behaviors of the TCP handshake. The first behavior illustrates a server repeatedly sending `SYN/ACK` segments, resulting in a TCP amplification. The second behavior points out a way for amplification even though the client (i.e., the victim) sends a `RST` segment to tear down the (not-yet-existing) TCP connection. In principle, this instructs servers (i.e, amplifiers) to stop sending any further `SYN/ACK` segments. We measured if hosts obey to this behavior.

4.2 Measuring TCP Amplification

As a first step to estimate the scope of this problem, we measure how the TCP stacks implement retransmissions during the TCP handshake. We perform an Internet-wide `SYN` scan and record the replies for further analysis. Our scanner does not complete the handshakes (i.e., we do not send `ACK` segments). With this, we aim to mimic the behavior of a system under an amplification attack that did not initiate the TCP connection in question. If TCP segments arrive that do not belong to any (half-)open connection (such as the reflected `SYN/ACK` segments in our scenario), TCP stacks either i) ignore these segments, or ii) respond with a `RST` segment, asking the other side to abort the TCP connection.

A victim, however, might not able to respond with `RST` packets, e.g., when it is already suffering from overload. Similarly, an attacker does not necessarily need to steer her attack against assigned IP addresses as shown in Figure 5. That is, the attacker can target an unassigned IP address so that there is no host that responds with `RST` segments. As a result, the capacity of the victim's net-

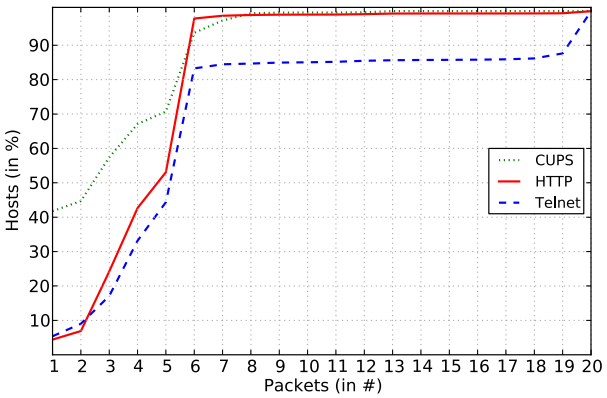


Figure 6: HTTP, Telnet, and CUPS without RST

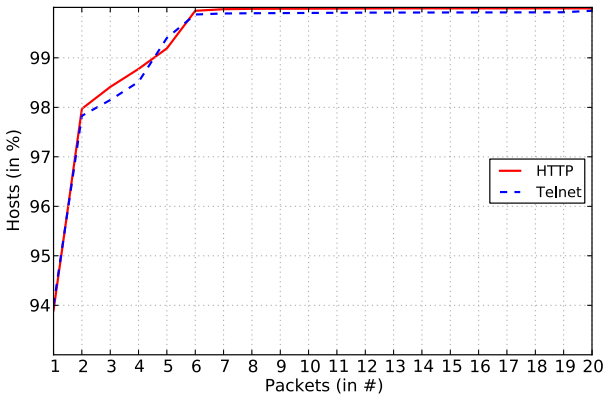


Figure 7: HTTP and Telnet with RST

work is overloaded by SYN/ACK segments. Addressing arbitrary IP addresses in the target network thus allows an attacker to control that no RST responses will be sent, leading to higher impact of an amplification attack.

TCP Scanning. We run two separated TCP scans to mimic both behaviors. In the first scan, we only send SYN segments and do not send anything back when receiving SYN/ACK segments. In the second scan, we "acknowledge" each incoming SYN/ACK segment with a RST segment. We performed the first scan for two popular protocols (i.e., HTTP and Telnet) and one printer-oriented protocol (CUPS). We chose these protocols, as according to Internet Census 2012, HTTP and Telnet yield a high number of reachable hosts [40]. In addition, to evaluate the TCP behavior of printers, we chose to scan CUPS.

In total, 66,785,451 HTTP hosts, 23,519,493 Telnet hosts, and 1,845,346 CUPS hosts replied to our requests. Figure 6 shows the results of the first scans as a CDF. The graph outlines that 6% of all HTTP hosts reply with a single SYN/ACK response and 24% of the hosts send at most three packets. A rise can be observed in-between 3 and 4 packets, meaning that 42% of the hosts reply to our SYN requests with 4 packets or less. We find the highest rise for 5 to 6 packets as 53.1% of all hosts send at most five SYN/ACK segments, but already 97.8% send six segments (or less). That is, 46.9% of the reachable HTTP hosts allow an amplification factor of 6 or higher.

Similar trends can be observed for the Telnet protocol. We find 55.6% of all Telnet hosts to enable an amplification attack with factor 6 or higher. In contrast to HTTP, about 12.2% of the Telnet hosts (more than 2.8 million systems) amplify requests even by factor 20. CUPS hosts, on the contrary, show less severe amplification rates. About 41.8% of the CUPS hosts replied with only a single packet and, in general, the number of segments sent by CUPS hosts is lower.

Figure 7 shows the results of our second scan (with RST). Indeed, sending RST tremendously changes the behavior of TCP stacks. In contrast to our first scan only 0.03% of Telnet hosts allow amplification by factor 20 which is a negligible number of 7,247 hosts. Almost 94% of all Telnet hosts send only one SYN/ACK packet in response; the same number applies for HTTP. A minor fraction of hosts keep resending SYN/ACK segments. We count 506,476 devices for HTTP and 114,157 devices for Telnet that send six SYN/ACK segments. Still, for the attack victim, replying with RST segments significantly reduces the impact for TCP amplification attacks and thus constitutes a potential proactive countermeasure.

4.3 Categorizing TCP Amplifiers

In this subsection, we aim to categorize the most prevalent behaviors that we have identified in the previous sections. Figures 6 and 7 have revealed significant groups of hosts that reply with the same number of TCP segments. Table 6 summarizes how many hosts belong to each of these groups. What kinds of systems are vulnerable to amplification attacks? To answer this question, we fingerprint the selected groups by re-using our TCP-based scans to obtain information via FTP, HTTP, HTTPS, SSH, and Telnet.

First, we classify the HTTP hosts that sent exactly six segments. Of the systems for which we obtained a fingerprint, about 88% (in total 3,228,000 hosts) run a Linux or Unix OS. Manual inspection has revealed that many of them are routers or embedded devices often running an FTP server (e.g., for a NAS). Other devices were vendor-specific, such as the ZynOS operating system. We also found numerous MikroTik devices and a smaller group of TP-LINK routers and D-Link devices. In contrast, there are only 0.2% Windows amplifiers in this group of hosts, hence this group of TCP amplifiers mainly consists of routers and various kinds of embedded devices.

Table 6: Hosts vulnerable to TCP amplification

	Number of response segments			
Scan	3	4	6	20
Without RST:				
CUPS	12.7%	9.7%	22.9%	0.0005%
	234,478	179,069	422,622	9
HTTP	17.3%	18.5%	44.7%	0.6%
	11,558,250	12,322,327	29,834,824	395,361
Telnet	8.1%	16.1%	38.9%	12.2%
	1,899,095	3,780,499	9,147,151	2,872,878
With RST:				
HTTP	0.44%	0.36%	0.76%	0.0005%
	294,535	243,028	506,476	349
Telnet	0.32%	0.37%	0.47%	0.03%
	76,774	88,504	114,157	7,247

In auxiliary tests, we have measured that Windows hosts (Windows 7, Vista, and XP) send four or five `SYN/ACK` segments in response—depending on the WINSOCK implementation. Although this amplification is not negligible, it is significantly lower than for other devices.

Second, we determine what kind of Telnet hosts sent six packets. Again, Unix and Linux are predominant as about 115 times more hosts in this group run Unix or Linux compared to the number of devices running Windows. Many of these hosts (49%) are routers, while other occurring devices are media servers, network cameras, digital video recorders, or VoIP phones.

Third, we analyze the Telnet hosts that sent 20 `SYN/ACK` segments. We found that 84.3% of all fingerprintable hosts in this group are routers or embedded devices. These embedded hosts—often based on MIPS or ARM architecture—include devices such as Raspberry Pis and printers. We found 86.1% of the devices to utilize the embedded web server *Allegro RomPager* and 37.5% to be manufactured by TP-LINK. In the remaining hosts, we also identified networking devices running ZynOS, ClearOS, or Cisco IOS. Typical desktop computers are negligible in this group: Windows is installed on 0.3% and MacOS on 0.0005% of these systems.

Lastly, we investigate the Telnet hosts that sent more than 20 `SYN/ACK` segments (21,981 hosts with an average of 971 response segments). Most of these hosts (87.9%) were found to be business and consumer routing devices of which 34% were running the embedded web server *GoAhead-Webs*. We find 50.4% of these devices to be a specific ATM Integrated Access device manufactured by RAD. Another 13.2% of the devices utilized the web server *Allegro RomPager* that we find to be associated to devices of manufacturers such as TP-LINK and ZyXEL. More information can be found in a further paper [20].

We conclude that amplification factors of 20 and more are largely caused by embedded devices and routers. We have contacted the vendors and wait for their feedback regarding these vulnerabilities.

5 Spoofer Identification

IP address spoofing is the root cause for amplification attacks, as it enables attackers to specify arbitrary targets that are flooded with reflected traffic. The Internet community addressed this issue as early as in May 2000 and suggested that—whenever possible—spoofed traffic should be blocked at the network edge [13]. However, as the attacks in practice have shown, spoofing still seems to be possible—yet it is unclear to what extent.

Up to now, the most powerful resource for tracking networks that allow spoofing is the Spoofer Project [36]. The project offers a client software that one can use to test if the own network filters IP-spoofed packets. Yet, such measurements require volunteers who download, compile/install, and run a client software. Aggregating user measurements in a study in July 2013, Beverly et al. show that about 610 of the 2582 tested ASes allow IP spoofing (at least partially, i.e., in some of their announced IP prefixes) [4]. On relative scale, however, less than 5% of the total number of ASes were tested. In other words, for more than 95% of the ASes it remains unclear if they support IP spoofing.

5.1 Remote Spoofer Test

Ideally, one would have a methodology to track networks that allow IP spoofing without the need for individuals running manual or tool-based tests from within the network. Such *remote tests* would boost the measurement coverage, so that we can alert administrators about potential misconfigurations that permit IP spoofing in their networks. We deploy such a large-scale experiment that enables us to identify thousands of ASes that support IP spoofing—from remote. Our DNS-based technique was first mentioned by Mauch on the NANOG mailing list in August 2013 [16]. It relies on public DNS proxies (or DNS stub resolvers—we will refer to "proxy" in the following) that have a broken networking implementation.

Figure 8 describes the core idea of the technique. The party that wants to identify spoofers (i.e., us) controls an Internet-scale scanner S and a name server that is authoritative for a domain suffix d_{suf}. In our case the domain d_{suf} is `scan.syssec.rub.de`. Note that we do not have control over devices on the right hand side, i.e., the DNS resolver and the optional DNS proxy, respectively. In step (1), the scanner S sends a DNS `A` lookup for domain d to an open resolver P. The domain d uses d_{suf} as domain suffix, but is specifically crafted for each scanning target. That is, S encodes a hex-formatted IP address of the scanning target P in the domain. This allows us to tell from the DNS response to which IP address we have sent the corresponding DNS request. In addition, to avoid caching

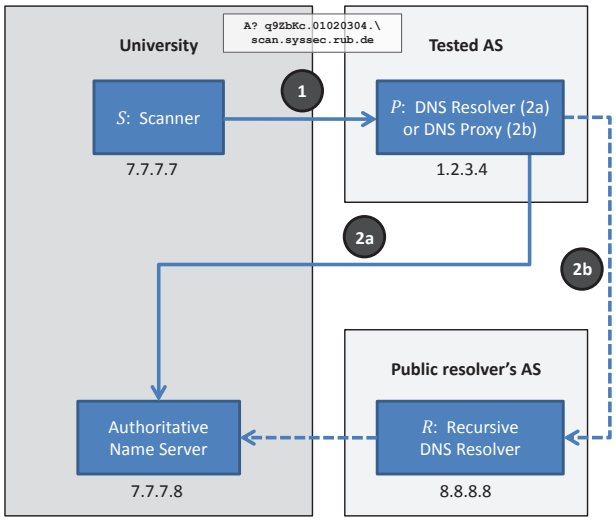


Figure 8: Setup to scan for spoofing-enabled networks. *R* is optional if *P* is an iterative resolver (step 2a) and is only used if *P* is a DNS proxy (step 2b)

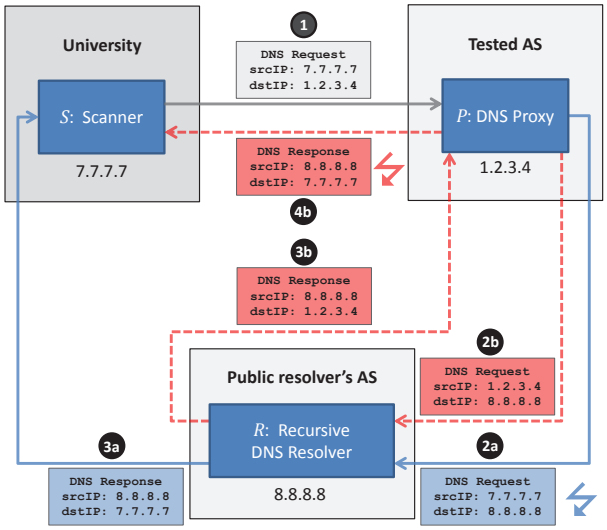


Figure 9: Network overview illustrating the two possible paths for DNS requests and responses when receiving responses for which the replying IP address did not match the IP address that was encoded in the requested domain

effects, we encode a random value in *d* that changes per request. In our case, *d* for target IP address 1.2.3.4 looks like `q9ZbKc.01020304.scan.syssec.rub.de`, whereas "q9ZbKc" is the random domain prefix and "01020304" the hex-formatted IP address.

When sending the request to *P*, one has to keep in mind that *P* may have different roles. If the scanned target *P* is a public recursive DNS resolver, *P* iteratively resolves the domain name by contacting the authoritative name servers down the domain tree as summarized in step (2a). For the purposes of our experiments such recursive resolvers are not important because they do not forward requests or responses. As we will show later, though, our technique to identify spoofing-enabled networks is based on the assumption that systems forward requests or responses. Quite often, *P* is not a resolver but a DNS proxy that forwards the DNS communication from a client (i.e., our scanner) to an iterative resolver *R*, as illustrated in step (2b).

We now leverage the fact that some DNS proxies do not correctly change the IP addresses when forwarding the request. In principle, to forward the DNS lookup to the resolver, the proxy *P* needs to change both the source and destination IP address of the request: it switches the source from *S* to its own address and the destination from its own address to *R*. Similarly, to forward the DNS response to the client, *P* changes the source from *R* to *P* and the destination from *P* to *S*. However, we encountered DNS proxies that do not change the addresses correctly. That is, we received DNS responses for which the replying IP address did not match the IP address that was encoded in the requested domain *d*. Instead, when *S* receives the response, the source IP address is set to *R*.

There are a few potential explanations for this observation. One is that *P* is a multi-homed system, i.e., has multiple interfaces with IP addresses in different networks. In many cases, though, proxies were using well-known resolvers (such as Google DNS) or resolvers in different ASes, excluding this possibility. Another explanation is that these devices have broken networking implementations, which cause the packets to have "wrong" IP header information when being forwarded. This could, e.g., be caused by broken Network Address Translation (NAT) implementations or faulty DNS proxy software.

Figure 9 illustrates the corresponding network situation when we receive a DNS response for which the source IP address of the UDP packet does not match the IP address that we encoded in the domain name of the DNS request. When sending a DNS request to the proxy (step 1), either *P* does not change the source address when forwarding the request to resolver *R* as outlined in step (2a) (i.e., the proxy effectively impersonates the sender *S*) so that *R* directly responds to *S* (step 3a). Alternatively, the proxy forwards the request to the resolver *R* (step 2b), obtains a DNS response (step 3b), and does not change the source address when forwarding the response to the sender *S* (step 4b), thus impersonates the resolver *R*. Either way, if *R* and *S* are not within *P*'s AS, then the proxy *P* is located in a network that permits the transmission of spoofed IP addresses. Both behaviors cause typical DNS clients to fail the resolution, as the DNS response comes from an unexpected IP address.

Table 7: Number of misconfigured DNS resolvers P and the corresponding Autonomous Systems

Filter	$\#P$	$\#AS_P$	Top 3 Countries
Top 4 Resolver	42,691	301	BR (52%), IT (11%), HU (10%)
Top 10 Resolver	45,072	352	BR (53%), IT (10%), HU (9%)
Distinct AS	170,451	2,692	CN (55%), BR (17%), RU (5%)
Distinct AS / $\#\text{resp}_R > 1$	161,988	2,063	CN (55%), BR (18%), RU (5%)
Distinct AS / $\#\text{resp}_R \geq 10$	137,075	870	CN (53%), BR (20%), RU (6%)

Clients will not even receive the replies if their network is protected by a stateful firewall, which drop UDP packets unrelated to any UDP stream known to the firewall. Unfortunately, we could not examine in detail which part of the forwarding was broken, as we did not control any of the recursive resolvers that the spoofing proxies used.

5.2 Finding Spoofing-Enabled Networks

While performing our Internet-wide scans, we observed a mismatch of source IP address and encoded target address for more than 2.2% of all responsive DNS servers, resulting in a total of 581,777 DNS proxies which redirect incoming requests to 225,888 distinct recursive DNS resolvers. To explore these misbehaving DNS proxies and the corresponding ASes in more detail, we enumerate the number of ASes permitting IP address spoofing using the following filtering methods:

(i) Our most conservative estimation is based only on responses from four commonly-used open resolvers operated by Google (i.e., 8.8.8.8 and 8.8.4.4) and OpenDNS (208.67.222.222 and 208.67.220.220). These servers ("Top 4") are a subset of the servers in the second approach.

(ii) Less conservative, we take into account DNS responses of the most popular ten resolvers ranked by the number of proxies using them ("Top 10").

(iii) Lastly, we focus on proxies for which $AS(S) \neq AS(P) \neq AS(R)$ applies. In other words, the proxy is not located in the same AS as both the sender S and the resolver R, and thus is spoofing the identity of one of these identities.

Table 7 illustrates the results obtained for each filtering method. In total, 7.7% of the potentially-spoofing DNS proxies forward the DNS requests to the Top 10 well-known resolvers (filter (ii)), resulting in 352 distinct ASes the proxies are located in. When limiting our focus to the Top 4 resolvers (filter (i)), we still identify 301 different ASes that permit spoofed traffic. Furthermore, we find 29.3% of all proxies to be located in different ASes than the sender S and the resolvers R (filter (iii)), resulting in 2,692 ASes permitting the proxies to either spoof the IP address of S or R.

Of the 225,888 individual resolvers R we find 50.7% utilized by multiple DNS proxies. To exclude potentially multi-homed systems with multiple interfaces in distinct ASes, we restrict the set of resolvers to those which responded to requests from multiple proxies and find 2,063 ASes that allow spoofed traffic. When further filtering the set to resolvers that replied to at least ten different proxies, we still identify 870 ASes permitting spoofing. Using our remote test, we can thus identify more spoofing-enabled ASes than the current state-of-the-art manual analyses performed by the Spoofer Project [36].

5.3 Fingerprinting IP-Spoofing Devices

Lastly, we want to understand what type of devices follow the weird practice of spoofing IP addresses while forwarding DNS requests/responses. For this, we use our TCP-based fingerprints to classify the 42,691 devices that used Google DNS or OpenDNS as iterative resolver. Of these devices, 6,120 devices replied to our TCP requests and 5,674 resolvers provided information suitable for fingerprinting. In total, we find 3,033 devices running the Dropbear SSH daemon, particularly employed on embedded devices. We also identify 1,437 MikroTik routers to be forwarding requests specifically to the Google DNS servers. Further 540 devices of the manufacturer Airlive perform similar behavior.

We achieve similar results when fingerprinting the hosts of the other filtering methods (see previous subsection). We again find Dropbear, MikroTik, and Airlive to appear frequently. We assume that these devices have either bad NAT rules or erroneous DNS proxy implementations. However, requests for more specific information from the vendors remained unanswered until now.

5.4 Remote Test Limitations

Our results show that DNS-based spoofing tests are a powerful resource to identify spoofing-enabled networks. One inherent limitation of this approach is, though, that such tests do only reveal the fact *that* (and not *if*) a network allows IP spoofing. We leave it up to future work to test if the tests can be expanded accordingly. For example, we could scan for DNS proxies that can be fingerprinted as systems that typically spoof IP addresses. In addition, collaborating with the recursive resolvers (such as OpenDNS or Google DNS) may reveal further insights about the spoofing systems. Lastly, given the large number of hosts running other protocols than DNS, it may be possible to use further protocols for similar remote spoofing tests.

6 Related Work

Our work was inspired by the analysis of amplification attacks by Rossow [33]. He identified 14 UDP-based network protocols that are vulnerable to amplification attacks and gave a thorough overview of countermeasures. We continue this line of research and classify the amplifiers. We show in an Internet-wide NTP amplifier notification initiative that the threats can be mitigated by cooperation within the security community. We furthermore investigate to what extent TCP-based amplification attacks are possible. Lastly, we provide an overview of spoofing-enabled networks. Our work is thus a thorough and novel extension of Rossow's initial analysis.

We group further related works by their topic:

TCP Amplification. To the best of our knowledge, we are the first to evaluate the amplification potential of the TCP three-way handshake. Prior work on TCP amplification has addressed guessable TCP sequence numbers, which in principle allow to establish TCP connections with spoofed packets [2, 31]. In addition, Paxson et al. looked at amplification in Transactional TCP (T/TCP)—which has very low popularity though [31]. Lastly, well-known stateful TCP attacks like the FTP bounce attack also allow for amplification [6]. Many of these attacks have been largely fixed with secure TCP stack implementations or by hardening certain protocols (e.g., FTP). The amplification vulnerabilities that we discovered in the TCP three-way handshake may again require improvements to TCP stacks.

Internet-Wide Scanning. Durumeric et al. presented ZMap, a publicly-available tool optimized for Internet-wide scans [11]. In fact, we leverage most of their proposed techniques and implemented their guidelines also for our custom scanner. Zhang et al. used Internet-wide scans to correlate the mismanagement and the maliciousness of networks [44]. They find networks that host open recursive DNS resolvers highly correlate to other malicious activities (such as spamming) initiated from these networks. Our work is orthogonal, as we follow a proactive approach to cooperate with the providers in order to get the vulnerabilities fixed. Two non-academic projects deployed by Mauch, the OpenNTPProject [30] and Open Resolver Project [29], also address the problems of amplification sources from a practical point of view. We have collaborated with Mauch to inform administrators of NTP servers vulnerable to the `monlist` amplification and are grateful for his support.

DDoS Attack Types. An alternative way to launch powerful DDoS attacks are networks of remotely-controllable bots. Büscher and Holz analyze DirtJumper, a botnet family with the specific task to perform DDoS attacks by abusing the Internet connection of infected desktop computers [5]. The DirtJumper botnet attacks at the application-level layer and does not aim to exhaust bandwidth, though. Kang et al. propose the Crossfire attack, in which bots direct low-intensity flows to a large number of publicly accessible servers [19]. These flows are concentrated on carefully chosen links such that they flood these links and disconnect target servers from the Internet. Studer and Perrig describe the Coremelt attack, in which bots send legitimate traffic to each other to flood and disable a network link between them [38]. All these attacks rely on bots, while our threat model only assumes that an attacker has any spoofing-enabled Internet uplink. Although the amplification DDoS attacks primarily try to congest bandwidth of a single victim, they can possibly be combined with the aforementioned techniques.

7 Conclusion

We have confirmed that amplification attacks remain a major Internet security issue—not only for UDP-based protocols. We identified TCP as an alternative source for amplification—despite its three-way-handshake protocol. We find millions of systems with TCP stacks that can be abused to amplify TCP traffic by a factor of 20x or higher. Our work revealed a tremendous number of potential amplification sources for both UDP and TCP-based protocols and classified these systems. During a first-ever large-scale notification campaign, we have observed a significant decrease in the number of amplifiers for NTP, giving hope for future attempts in fixing protocols that have similarly-severe amplification vulnerabilities. Finally, our remote spoofing test has identified more than 2,000 networks that do not use proper egress filtering—indicating that it is still a long way to go until we will have a spoofing-free Internet.

Acknowledgment

We would like to thank the anonymous reviewers and Jared Mauch for their constructive and valuable comments. This work was supported by the German Federal Ministry of Education and Research (BMBF Grants 16BY1110 (MoBE) and 16BY1201D (iAID)).

References

[1] BAILEY, M., COOKE, E., JAHANIAN, F., AND NAZARIO, J. The Internet Motion Sensor - A Distributed Blackhole Monitoring System. In *Symposium on Network and Distributed System Security (NDSS)* (2005).

[2] BELLOVIN, S. RFC 1984: Defending Against Sequence Number Attacks, 1996.

[3] BEVERLY, R., BERGER, A., HYUN, Y., AND CLAFFY, K. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Internet Measurement Conference (IMC)* (2009).

[4] BEVERLY, R., KOGA, R., AND CLAFFY, K. Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet. *Internet Society* (2013).

[5] BÜSCHER, A., AND HOLZ, T. Tracking DDoS Attacks: Insights into the Business of Disrupting the Web. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2012).

[6] CENTER, C. C. FTP Bounce: Advisory CA-1997-27. `http://www.cert.org/advisories/CA-1997-27.html`, 1997.

[7] CISCO PSIRT. Cisco Network Time Protocol Distributed Reflective Denial of Service Vulnerability. `http://tools.cisco.com/security/center/content/CiscoSecurityNotice/CVE-2013-5211`, 2014.

[8] COMPUTER EMERGENCY RESPONSE TEAM. CERT advisory CA-1996-21: TCP SYN Flooding and IP Spoofing Attacks. `https://www.cert.org/historical/advisories/CA-1996-21.cfm`, 1996.

[9] DITTRICH, D. Distributed Denial of Service (DDoS) Attacks/tools. `http://staff.washington.edu/dittrich/misc/ddos/`, 2000.

[10] DURUMERIC, Z., BAILEY, M., AND HALDERMAN, J. A. An Internet-wide View of Internet-wide Scanning. In *USENIX Security Symposium* (2014).

[11] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium* (2013).

[12] EHRENKRANZ, T., AND LI, J. On the State of IP Spoofing Defense. *ACM Trans. Internet Technol. 9*, 2 (May 2009).

[13] FERGUSON, P., AND SENIE, D. BCP 38: Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing, 1998.

[14] FREILING, F. C., HOLZ, T., AND WICHERSKI, G. Botnet Tracking: Exploring a Root-cause Methodology to Prevent Distributed Denial-of-service Attacks. In *European Symposium on Research in Computer Security (ESORICS)* (2005).

[15] IOANNIDIS, J., AND BELLOVIN, S. M. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2002).

[16] J. MAUCH. `http://seclists.org/nanog/2013/Aug/132`, August 2013.

[17] JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *World Wide Web Conference (WWW)* (2002).

[18] KANDULA, S., KATABI, D., JACOB, M., AND BERGER, A. Botz-4-sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In *USENIX Symposium on Networked Systems Design and Implementation* (2005).

[19] KANG, M. S., LEE, S. B., AND GLIGOR, V. D. The Crossfire Attack. In *Proceedings of IEEE Security and Privacy (S&P)* (San Francisco, CA, USA, 2013).

[20] KÜHRER, M., HUPPERICH, T., ROSSOW, C., AND HOLZ, T. Hell of a Handshake: Abusing TCP for Reflective Amplification DDoS Attacks. In *USENIX Workshop on Offensive Technologies (WOOT)* (2014).

[21] M. PRINCE; CLOUDFLARE, INC. `http://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet`, March 2013.

[22] M. PRINCE; CLOUDFLARE, INC. `http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack`, February 2014.

[23] MAXMIND GEOIP DATABASE. `http://www.maxmind.com/en/ip-location`, 2014.

[24] MIRKOVIC, J., DIETRICH, S., DITTRICH, D., AND REIHER, P. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall PTR, 2004.

[25] MITRE. CVE-2013-5211. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5211`, 2013.

[26] MOORE, D., VOELKER, G. M., AND SAVAGE, S. Inferring Internet Denial-of-service Activity. In *USENIX Security Symposium* (2001).

[27] NETWORK MAPPER. `http://nmap.org/`, 2014.

[28] NTP POOL PROJECT. `http://www.pool.ntp.org`, 2014.

[29] OPEN RESOLVER PROJECT. `http://OpenResolverProject.org/`, 2013.

[30] OPENNTP SCANNING PROJECT. `http://OpenNTPProject.org/`, 2014.

[31] PAXSON, V. An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks. In *Computer Communication Review 31(3)* (July 2001).

[32] PORRAS, PHILLIP AND SAIDI, HASSEN AND YEGNESWARAN, VINOD. Technical Report: Conficker C Analysis. `http://mtc.sri.com/Conficker/`, 2009.

[33] ROSSOW, C. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[34] SCHUBA, C. L., KRSUL, I. V., KUHN, M. G., SPAFFORD, E. H., SUNDARAM, A., AND ZAMBONI, D. Analysis of a Denial of Service Attack on TCP. In *IEEE S&P* (1997).

[35] SHADOWSERVER FOUNDATION. `http://shadowserver.org/`, 2014.

[36] SPOOFER PROJECT. `http://spoofer.cmand.org/`, 2014.

[37] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[38] STUDER, A., AND PERRIG, A. The Coremelt Attack. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (Saint Malo, France, September 2009).

[39] SYMANTEC CORPORATION. Hackers Spend Christmas Break Launching Large Scale NTP-Reflection Attacks. `http://www.symantec.com/connect/blogs/hackers-spend-christmas-break-launching-large-scale-ntp-reflection-attacks`, 2013.

[40] THE INTERNET CENSUS 2012. Port scanning /0 using insecure embedded devices. `http://internetcensus2012.bitbucket.org/paper.html`, 2012.

[41] TRUSTED INTRODUCER. `http://trusted-introducer.org/`, 2014.

[42] US-CERT. NTP Amplification Attacks Using CVE-2013-5211. `http://www.us-cert.gov/ncas/alerts/TA14-013A`, 2014.

[43] YAAR, A., PERRIG, A., AND SONG, D. Pi: A Path Identification Mechanism to Defend Against DDoS Attacks. In *IEEE S&P* (2003).

[44] ZHANG, J., DURUMERIC, Z., BAILEY, M., LIU, M., AND KARIR, M. On the Mismanagement and Maliciousness of Networks. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport

Rob Jansen[†] *John Geddes*[‡] *Chris Wacek*[*] *Micah Sherr*[*] *Paul Syverson*[†]

[†] *U.S. Naval Research Laboratory*
{*rob.g.jansen, paul.syverson*}*@nrl.navy.mil*

[‡] *University of Minnesota*
geddes@cs.umn.edu

[*] *Georgetown University*
{*cwacek, msherr*}*@cs.georgetown.edu*

Abstract

Tor's growing popularity and user diversity has resulted in network performance problems that are not well understood. A large body of work has attempted to solve these problems without a complete understanding of where congestion occurs in Tor. In this paper, we first study congestion in Tor at individual relays as well as along the entire end-to-end Tor path and find that congestion occurs almost exclusively in egress kernel socket buffers. We then analyze Tor's socket interactions and discover two major issues affecting congestion: Tor writes sockets sequentially, and Tor writes as much as possible to each socket. We thus design, implement, and test KIST: a new socket management algorithm that uses real-time kernel information to *dynamically compute the amount to write* to each socket while considering *all writable circuits* when scheduling new cells. We find that, in the medians, KIST reduces circuit congestion by over 30 percent, reduces network latency by 18 percent, and increases network throughput by nearly 10 percent. We analyze the security of KIST and find an acceptable performance and security trade-off, as it does not significantly affect the outcome of well-known latency and throughput attacks. While our focus is Tor, our techniques and observations should help analyze and improve overlay and application performance, both for security applications and in general.

1 Introduction

Tor [21] is the most popular overlay network for communicating anonymously online. Tor serves millions of users daily by transferring their traffic through a source-routed *circuit* of three volunteer relays, and encrypts the traffic in such a way that no one relay learns both its source and intended destination. Tor is also used to resist online censorship, and its support for hidden services, network bridges, and protocol obfuscation has helped attract a large and diverse set of users.

While Tor's growing popularity, variety of use cases, and diversity of users have provided a larger anonymity set, they have also led to performance issues [23]. For example, it has been shown that roughly half of Tor's traffic can be attributed to BitTorrent [18, 43], while the more recent use of Tor by a botnet [29] has further increased concern about Tor's ability to utilize volunteer resources to handle a growing user base [20, 36, 37, 45].

Numerous proposals have been made to battle Tor's performance problems, some of which modify the mechanisms used for path selection [13, 59, 60], client throttling [14, 38, 45], circuit scheduling [57], and flow/congestion control [15]. While some of this work has or will be incorporated into the Tor software, none of it has provided a comprehensive understanding of *where* the most significant source of congestion occurs in a complete Tor deployment. This lack of understanding has led to the design of uninformed algorithms and speculative solutions. In this paper, we seek a more thorough understanding of congestion in Tor and its effect on Tor's security. We explore an answer to the fundamental question—"*Where* is Tor slow?"—and design informed solutions that not only decrease congestion, but also improve Tor's ability to manage it as Tor continues to grow.

Congestion in Tor: We use a multifaceted approach to exploring congestion. First, we develop a shared library and Tor software patch for measuring congestion *local to relays* running in the public Tor network, and use them to measure congestion from three live relays under our control. Second, we develop software patches for Tor and the open-source Shadow simulator [7], and use them to measure congestion along the *full end-to-end path* in the largest known, at-scale, private Shadow-Tor deployment. Our Shadow patches ensure that our congestion measurements are accurate and realistic; we show how they significantly improve Shadow's TCP implementation, network topology, and Tor models.[1]

[1] We have contributed our patches to the Shadow project [7] and they have been integrated as of Shadow release `1.9.0`.

To the best of our knowledge, we are the first to consider such a comprehensive range of congestion information that spans from individual application instances to full network sessions for the entire distributed system. Our analysis indicates that congestion occurs almost exclusively inside of the kernel egress socket buffers, dwarfing the Tor and the kernel ingress buffer times. This finding is consistent among all three public Tor relays we measured, and among relays in every circuit position in our private Shadow-Tor deployment. This result is significant, as Tor does not currently prevent, detect, or otherwise manage kernel congestion.

Mismanaged Socket Output: Using this new understanding of *where* congestion occurs, we analyze Tor's socket output mechanisms and find two significant and fundamental design issues: Tor *sequentially* writes to sockets while ignoring the state of all sockets other than the one that is currently being written; and Tor writes *as much as possible* to each socket.

By writing to sockets sequentially, Tor's circuit scheduler considers only a small subset of the circuits with writable data. We show how this leads to improper utilization of circuit priority mechanisms, which causes Tor to send lower priority data from one socket *before* higher priority data from another. This finding confirms evidence from previous work indicating the ineffectiveness of circuit priority algorithms [35].

By writing as much as possible to each socket, Tor is often delivering to the kernel more data than it is capable of sending due to either physical bandwidth limitations or throttling by the TCP congestion control protocol. Not only does writing too much increase data queuing delays in the kernel, it also further reduces the effectiveness of Tor's circuit priority mechanisms because Tor relinquishes control over the priority of data after it is delivered to the kernel.[2] This kernel overload is exacerbated by the fact that a Tor relay may have thousands of sockets open at any time in order to facilitate data transfer between other relays, a problem that may significantly worsen if Tor adopts recent proposals [16, 26] that suggest increasing the number of sockets between relays.

KIST: Kernel-Informed Socket Transport: To solve the socket management problems outlined above, we design KIST: a Kernel-Informed Socket Transport algorithm. KIST has two features that work together to significantly improve Tor's control over network congestion. First, KIST changes Tor's circuit level scheduler so that it chooses from *all* circuits with writable data rather than just those belonging to a single TCP socket. Second, to complement this global scheduling approach, KIST also dynamically manages the amount of data written to each socket based on real-time kernel and TCP state information. In this way, KIST attempts to minimize the amount of data that exists in the kernel that cannot be sent, and to maximize the amount of time that Tor has control over data priority.

We perform in-depth experiments in our at-scale private Shadow-Tor network, and we show how KIST can be used to relocate congestion from the kernel into Tor, where it can be properly managed. We also show how KIST allows Tor to correctly utilize its circuit priority scheduler, reducing download latency by over 660 milliseconds, or 23.5 percent, for interactive traffic streams typically generated by web browsing behaviors.

We analyze the security of KIST, showing how it affects well-known latency and throughput attacks. In particular, we show the extent to which the latency improvements reduce the number of round-trip time measurements needed to conduct a successful latency attack [31]. We also show how KIST does not significantly affect an adversary's ability to collect accurate measurements required for the throughput correlation attack [44] when compared to vanilla Tor.

Outline of Major Contributions: We outline our major contributions as follows:

- in Section 3 we discuss improvements to the open-source Shadow simulator that significantly enhance its accuracy, including experiments with the largest known private Tor network of 3,600 relays and 13,800 clients running real Tor software;
- in Section 4 we discuss a library we developed to measure congestion in Tor, and results from the first known end-to-end Tor circuit congestion analysis;
- in Section 5 we show how Tor's current management of sockets results in ineffective circuit priority, detail the KIST design and prototype, and show how it improves Tor's ability to manage congestion through a comprehensive and full-network evaluation; and
- in Section 6 we analyze Tor's security with KIST by showing how our performance improvements affect well-known latency and throughput attacks.

2 Background and Related Work

Tor [21] is a volunteer-operated anonymity service used by an estimated hundreds of thousands of daily users [28]. Tor assumes an adversary who can monitor a portion of the underlying Internet and/or operate Tor relays. People primarily use Tor to prevent an adversary from discovering the endpoints of their communications, or disrupting access to information.

Tor Traffic Handling: Tor provides anonymity by forming source-routed paths called *circuits* that consist of (usually) three relays on an overlay network. Clients transfer TCP-based application traffic within these circuits; encrypted application-layer headers and payloads

[2] To the best of our knowledge, the Linux kernel uses a variant of the first-come first-serve queuing discipline among sockets.

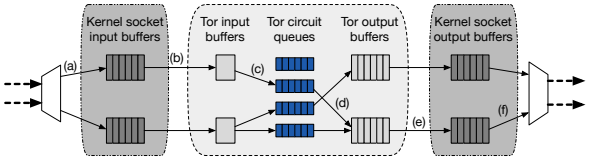


Figure 1: Internals of cell processing within in a Tor relay. Dashed lines denote TCP connections. Transitions between buffers—both within the kernel (side boxes) and within Tor (center box)—are shown with solid arrows.

make it more difficult for an adversary to discern an intercepted communication's endpoints or learn its plaintext.

A given circuit may carry several Tor *streams*, which are logical connections between clients and destinations. For example, a HTTP request to usenix.org may result in several Tor streams (i.e., to fetch embedded objects); these streams may all be transported over a single circuit. Circuits are themselves multiplexed over TLS connections between relays whenever their paths share an edge; that is, all concurrent circuits between relays u and v will be transferred over the same TLS connection between the two relays, irrespective of the circuits' endpoints.

The unit of transfer in Tor is a 512-byte *cell*. Figure 1 depicts the internals of cell processing within a Tor relay. In this example, the relay maintains two TLS connections with other relays. Incoming packets from the two TCP streams are first demultiplexed and placed into kernel socket input buffers by the underlying OS (Figure 1a)[3]. The OS processes the packets, usually in FIFO order, delivering them to Tor where they are reassembled into TLS-encrypted cells using dedicated Tor input buffers (Figure 1b). Upon receipt of an entire TLS datagram, the TLS layer is removed, the cell is onion-crypted,[4] and then transferred and enqueued in the appropriate Tor circuit queue (Figure 1c). Each relay maintains a queue for each circuit that it is currently serving. Cells from the same Tor input buffer may be enqueued in different circuit queues, since a single TCP connection between two relays may carry multiple circuits.

Tor uses a priority-based circuit scheduling approach that attempts to prioritize interactive web clients over bulk downloaders [57]. The circuit scheduler selects a cell from a circuit queue to process based on this prioritization, onion-crypts the cell, and stores it in a Tor output buffer (Figure 1d). Once the Tor output buffer contains sufficient data to form a TLS packet, the data is written to the kernel for transport (Figure 1f).

Improving Tor Performance: There is a large body of work that attempts to improve Tor's network performance, e.g., by refining Tor's relay selection strategy [11,55,56] or providing incentives to users to operate Tor relays [36,37,45]. These approaches are orthogonal and can be applied in concert with our work, which focuses on improving Tor's congestion management.

Most closely related to this paper are approaches that modify Tor's circuit scheduling, flow control, or transport mechanisms. Reardon and Goldberg suggest replacing Tor's TCP-based transport mechanism with UDP-based DTLS [54], while Mathewson explores using SCTP [40]. Murdoch [47] explains that the UDP approach is promising, but there are challenges that have thus far prevented the approach from being deployed: there is limited kernel support for SCTP; and the lack of hop-by-hop reliability from UDP-based transports causes increased load at Tor's exit relays. Our work allows Tor to best utilize the existing TCP transport in the short term while work toward a long term UDP deployment strategy continues.

Tang and Goldberg propose the use of the exponential weighted moving average (EWMA) to characterize circuits' recent levels of activity, with bursty circuits given greater priority than busy circuits [57] (to favor interactive web users over bulk downloaders). Unfortunately, although Tor has adopted EWMA, the network has not significantly benefitted from its use [35]. In our study of *where* Tor is slow, we show that EWMA is made ineffective by Tor's current management of sockets, and can be made effective through our proposed modifications.

AlSabah et al. propose an ATM-like congestion and flow control system for Tor called N23 [15]. Their approach causes pushback effects to previous nodes, reducing congestion in the entire circuit. Our KIST strategy is complementary to N23, focusing instead on local techniques to remove kernel-level congestion at Tor relays.

Torchestra [26] uses separate TCP connections to carry interactive and bulk traffic, isolating the effects of congestion between the two traffic classes. Conceptually, Torchestra moves circuit-selection logic to the kernel, where the OS schedules packets for the two connections. Relatedly, AlSabah and Goldberg introduce PCTCP [16], a transport mechanism for Tor in which each circuit is assigned its own IPsec tunnel. In this paper, we argue that overloading the kernel with additional sockets reduces the effectiveness of circuit priority mechanisms since the kernel has no information regarding the priority of data. In contrast, we aim to move congestion management to Tor, where priority scheduling can be most effective.

Nowlan et al. [50] propose the use of uTCP and uTLS [49] to tackle the "head-of-line" blocking problem in Tor. Here, they bypass TCP's in-order delivery mechanism to peek at traffic that has arrived but is not ready to be delivered by the TCP stack (e.g., because an earlier packet was dropped). Since Tor multiplexes multiple cir-

[3]For simplicity, we consider only relays that run Linux since such relays represent 75% of all Tor relays and contribute 91% of the bandwidth of the live Tor network [58].

[4]Encrypted or decrypted, depending on circuit direction.

cuits over a single TCP connection, their technique offers significant latency improvements when connections are lossy, since already-arrived traffic can be immediately processed. Our technique can be viewed as a form of application-layer head-of-line countermeasure since we move scheduling decisions from the TCP stack to within Tor. In contrast to Nowlan et al.'s approach, we do not require any kernel-level modifications or changes to Tor's transport mechanism.

3 Enhanced Network Experimentation

To increase confidence in our experiments, we introduce three significant enhancements to the Shadow Tor simulator [35] and its existing models [33]: a more realistic simulated kernel and TCP network stack, an updated Internet topology model, and the largest known deployed private Tor network. The enhancements in this section represent a large and determined engineering effort; we will show how Tor experimental accuracy has significantly benefited as a result of these improvements. We remark that our improvements to Shadow will have an immediate impact beyond this work to the various research groups around the world that use the simulator.

Shadow TCP Enhancements: After reviewing Shadow [7], we first discovered that it was missing many important TCP features, causing it to be less accurate than desired. We enhanced Shadow by adding the following: retransmission timers [52], fast retransmit/recovery [12], selective acknowledgments [42], and forward acknowledgments [41]. Second, we discovered that Shadow was using a very primitive version of the basic additive-increase multiplicative-decrease (AIMD) congestion control algorithm. We implemented a much more complete version of the CUBIC algorithm [27], the default congestion control algorithm used in the Linux kernel since version 2.6.19. CUBIC is an important algorithm for properly adjusting the congestion window. We will show how our implementation of these algorithms greatly enhance Shadow's accuracy, which is paramount to the remainder of this paper. See Appendix A.1 [34] for more details about our modifications.

We verify the accuracy of Shadow's new TCP implementation to ensure that it is adequately handling packet loss and properly growing the congestion window by comparing its behavior to ns [51], a popular network simulator, because of the ease at which ns is able to model packet loss rates. In our first experiment, both Shadow and ns have two nodes connected by a 10 MiB/s link with a 10 ms round trip time. One node then downloads a 100 MiB file 10 times for each tested packet loss rate. Figure 2a shows that the average download time in Shadow matches well with ns over varying packet loss rates. Although not presented here, we similarly validated Shadow with our changes against a real network link using the bandwidth and packet loss rate that was achieved over our switch; the results did not significantly deviate from those presented in Figure 2a.

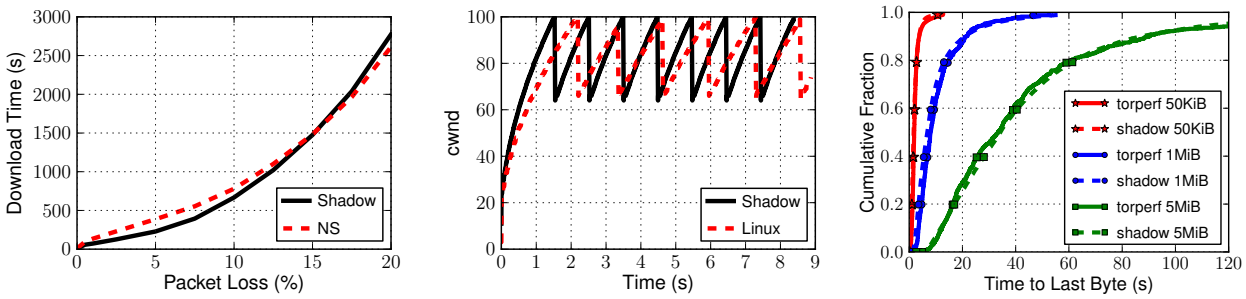
For our second experiment, we check that the growth of the congestion window using CUBIC is accurate. We first transfer a 100 MiB file over a 100 Mbit/s link between two physical Ubuntu 12.04 machines running the 3.2.0 Linux kernel. We record the `cwnd` (congestion window) and `ssthresh` (slow start threshold) values from the `getsockopt` function call using the `TCP_INFO` option. We then run an identical experiment in Shadow, setting the slow start threshold to what we observed from Linux and ensuring that packet loss happens at roughly the same rate. Figure 2b shows the value of `cwnd` in both Shadow and Linux over time, and we see almost identical growth patterns. The slight variation in the saw-tooth pattern is due to unpredictable variation in the physical link that was not reproduced by Shadow. As a result, Shadow's `cwnd` grew slightly faster than Linux's because Shadow was able to send one extra packet. We believe this is an artifact of our particular physical configuration and do not believe it significantly affects simulation accuracy in general: more importantly, the overall saw-tooth pattern matches well.

The two experiments discussed above give us high confidence that our TCP implementation is accurate, both in responding to packet loss and in operation of the CUBIC congestion control algorithm.

Shadow Topology Enhancements: To ensure that we are causing the most realistic performance and congestion effects possible during simulation, we enhance Shadow using techniques from recent research in modeling Tor topologies [39, 59], traceroute data from CAIDA [2], and client/server data from the Tor Metrics Portal [8] and Alexa [1]. This data-driven Internet map is more realistic than the one Shadow provides, and includes 699,029 vertices and 1,338,590 edges. For space reasons, we provide more details in Appendix A.2 [34].

Tor Model: Using Shadow with the improvements discussed above, we build a Tor model that reflects the real Tor network as it existed in July 2013, using the then-latest stable Tor version `0.2.3.25`. (We use this model for all experiments in this paper.) Using data from the Tor Metrics Portal [8], we configure a complete, private Tor network following Tor modeling best practices [33], and attach every node to the closest network location in our topology map. The resulting Tor network configuration includes 10 directory authorities, 3,600 relays, 13,800 clients, and 4,000 file servers—the largest known working private experimental Tor network, and the first to run at scale to the best of our knowledge.

The 13,800 clients in our model provide background traffic and load on the network. 10,800 of our clients



(a) Download Time, Shadow vs. ns (b) Congestion Window, Shadow vs. Linux (c) Download Time, Shadow-Tor vs. Tor

Figure 2: Figure 2a compares Shadow to ns download times. Figure 2b compares congestion window over time when Shadow and Linux have the same link properties. Figure 2c compares Shadow-Tor to live Tor measurements collected from Tor Metrics [8].

download 320 KiB files (the median size of a web page according to the most recent web statistics published by a Google engineer [53]) and then wait for a time chosen uniformly at random from the range [1, 60 000] milliseconds after each completed download. 1,200 of our clients repeatedly download 5 MiB files with no pauses between completing a download and starting the next. The ratio of these client behaviors was chosen according to the latest known measurements of client traffic on Tor [18, 43]. Shadow also contains 1,800 TorPerf [9] clients that download a file over a fresh circuit and pause for 60 seconds after each successful download. (TorPerf is a tool for measuring Tor performance.) 600 of the TorPerf clients download 50 KiB files, 600 download 1 MiB files, and 600 download 5 MiB files. Our simulations run for one virtual hour during each experiment.

Figure 2c shows a comparison of publicly available TorPerf measurements collected on the live Tor network [8] to those collected in our private Shadow-Tor network. As shown in Figure 2c, our full size Shadow-Tor network is extremely accurate in terms of time to complete downloads for all file sizes. These results give us confidence that our at-scale Shadow-Tor network is strongly representative of the deployed Tor network.

4 Congestion Analysis

In this section, we explore *where* congestion happens in Tor through a large scale congestion analysis. We take a multifaceted approach by measuring congestion as it occurs in both the live, public Tor network, and in an experimental, private Tor network running in Shadow. By analyzing relays in the public Tor network, we get the most realistic and accurate view of what is happening at our measured relays. We supplement the data from a relatively small public relay sample with measurements from a much larger set of private relays, collecting a larger and more complete view of Tor congestion.

To understand congestion, we are interested in measuring the time that data spends inside of Tor as well as inside of kernel sockets in both the incoming and outgoing directions. We will discuss our findings in both environments after describing the techniques that we used to collect the time spent in these locations.

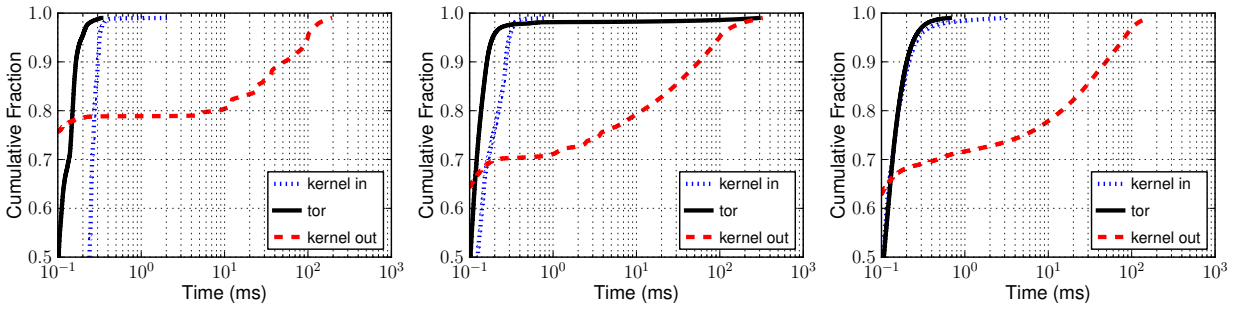
4.1 Congestion in the Live Tor Network

Relays running in the operational network provide the most accurate source of congestion data, as these relays are serving real clients and transferring real traffic. As mentioned above, we are interested in measuring queuing times inside of the Tor application as well as inside of the kernel, and so we developed techniques for both in the local context of a public Tor relay.

Tor Congestion: Measuring Tor queuing times requires some straightforward modifications to the Tor software. As soon as a relay reads the entire cell, it internally creates a cell structure that holds the cell's circuit ID, command, and payload. We add a new unique cell ID value. Whenever a cell enters Tor and the cell structure is created, we log a message containing the current time and the cell's unique ID. The cell is then switched to the outgoing circuit. After it's sent to the kernel we log another message containing the time and ID. The difference between these times represents Tor application congestion.

Kernel Congestion: Measuring kernel queuing times is much more complicated since Tor does not have direct access to the kernel internals. In order to log the times when a piece of data enters and leaves the kernel in both the incoming and outgoing directions, we developed a new, modular, application-agnostic, multithreaded library, called `libkqtime`.[5] `libkqtime` uses `libpcap` [6] to determine when data crosses the host/network boundary, and *function interposition* on

[5] `libkqtime` was written in 770 LOC, and is available for download as open source software [5].



(a) curiosity1 (b) curiosity2 (c) curiosity3

Figure 3: The distribution of congestion inside Tor and the kernel on our 3 relays running in the public Tor network, as measured between 2014-01-20 and 2014-01-28. Most congestion occurred in the outbound kernel queues on all three relays.

the `write()`, `send()`, `read()`, and `recv()` functions to determine when it crosses the application/kernel boundary. The library copies a 16 byte tag as data enters the kernel from either end, and then searches for the tag as data leaves the kernel on the opposite end. This process works in both directions, and the timestamps collected by the library allow us to measure both inbound and outbound kernel congestion. Appendix B [34] gives a more detailed description of `libkqtime`.

Results: To collect congestion information in Tor, we first ran three live relays (curiosity1, curiosity2, and curiosity3) using an unmodified copy of Tor release `0.2.3.25` for several months to allow them to stabilize. We configured them as non-exit nodes and used a network appliance to rate limit curiosity1 at 1 Mbit/s, curiosity2 at 10 Mbit/s, and curiosity3 at 50 Mbit/s. Only curiosity2 had the guard flag (could be chosen as entry relay for a circuit) during our data collection. On 2014-01-20, we swapped the Tor binary with a version linked to `libkqtime` and modified as discussed in Section 4.1. We collected Tor and kernel congestion for 190 hours (just under 8 days) ending on 2014-01-28, and then replaced the vanilla Tor binary.

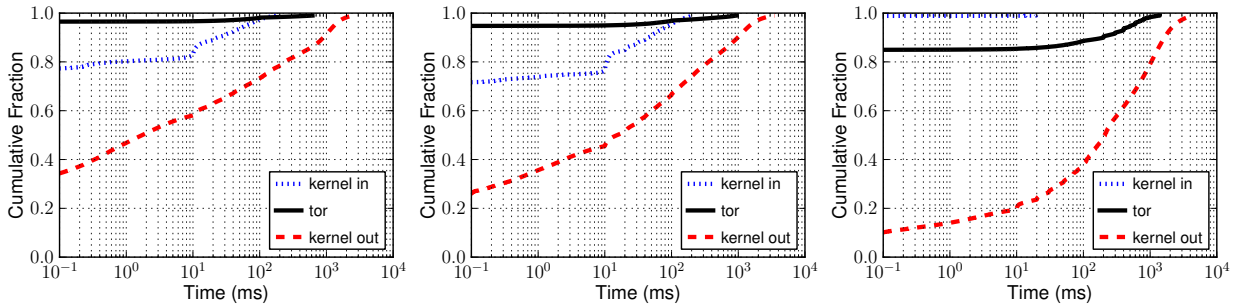
The distributions of congestion as measured on each relay during the collection period are shown in Figure 3 with logarithmic x-axes. Our measurements indicate that most congestion, when present, occurs in the *kernel outbound queues*, while kernel inbound and Tor congestion are both less than 1 millisecond for over 95 percent of our measurements. This finding is consistent across all three relays we measured. Kernel outbound congestion increases from curiosity1 to curiosity2, and again slightly from curiosity2 to curiosity3, indicating that congestion is a function of relay capacity or load. We leave it to future work to analyze the strength of this correlation, as that is outside the scope of this paper.

Ethical Considerations: We took careful protections to ensure that our live data collection did not breach users' anonymity. In particular, *we captured only buffered data timing information*; no network addresses were ever recorded. We discussed our experimental methodology with Tor Project maintainers, who raised no objections. Finally, we contacted the IRB of our relay host institution. The IRB decided that no review was warranted since our measurements did not, in their opinion, constitute human subjects research.

4.2 Congestion in a Shadow-Tor Network

While congestion data from real live relays is the most accurate, it only gives us a limited view of congestion local to our relays. The congestion measured at our relays may or may not be representative of congestion at other relays in the network. Therefore, we use our private Shadow-Tor network to supplement our congestion data and enhance our analysis. Using Shadow provides many advantages over live Tor: it's technically simpler; we are able to measure congestion *at all relays* in our private network; we can track the congestion of every cell *across the entire circuit* because we do not have privacy concerns with Shadow; and we can analyze how congestion changes with varying network configurations.

Tor and Kernel Congestion: The process for collecting congestion in Shadow is simpler than in live Tor, since we have direct access to Shadow's virtual kernel. In our modified Tor, each cell again contains a unique ID as in Section 4.1. However, when running in Shadow, we also add a 16 byte magic token and include both the unique ID and the magic token when sending cells out to the network. The unique ID is forwarded with the cell as it travels through the circuit. Since Shadow prevents Tor from encrypting cell contents for efficiency reasons, the Shadow kernel can search outgoing packets for the unencrypted magic token immediately before they leave the virtual network interface. When found, it logs the unique cell ID with a timestamp. It performs an analogous procedure for incoming packets immediately after they arrive on the virtual network interface. These



(a) Relays in Entry Position (b) Relays in Middle Position (c) Relays in Exit Position

Figure 4: Relay congestion by circuit position in our Shadow-Tor network, measured on on circuits using end-to-end cells from 1,200 clients selected uniformly at random. Most congestion occurred in the outbound kernel queues, independent of relay position.

Shadow timestamps are combined with the timestamps logged when a cell enters and leaves Tor to compute both Tor and kernel congestion.

Results: We use our model of Tor as described in Section 3, with the addition of the cell tracking information discussed above. Since tracking every cell would consume an extremely large amount of disk space, we sample congestion as follows: we select 10 percent of the non-TorPerf clients (1,200 total) in our network chosen uniformly, and track 1 of every 100 cells traveling over circuits they initiate. The tracking timestamps from these cells are then used to attribute congestion to the relays through which the cells are traveling.

It is important to understand that our method does not sample *relay* congestion uniformly: the congestion measurements will be biased towards relays that are chosen more often by clients, according to Tor's bandwidth-weighted path selection algorithm. This means that our results will represent the congestion that a typical *client* will experience when using Tor. We believe that these results are more meaningful than those we could obtain by uniformly sampling congestion at each relay independently (as we did in Section 4.1), because ultimately we are interested in improving clients' experience.

The distributions of congestion measured in Shadow for each circuit position are shown in Figure 4. We again find that congestion occurs most significantly in the kernel outbound queues, regardless of a relay's circuit position. Our Shadow experiments indicate higher congestion than in live Tor, which we attribute to our client-oriented sampling method described above.

5 Kernel-Informed Socket Transport

Our large scale congestion analysis from Section 4 revealed that the most significant delay in Tor occurs in outbound kernel queues. In this section, we first explore how this problem adversely affects Tor's traffic management by disrupting existing scheduling mechanisms to the extent that they become ineffective. We then describe the KIST algorithm and experimental results.

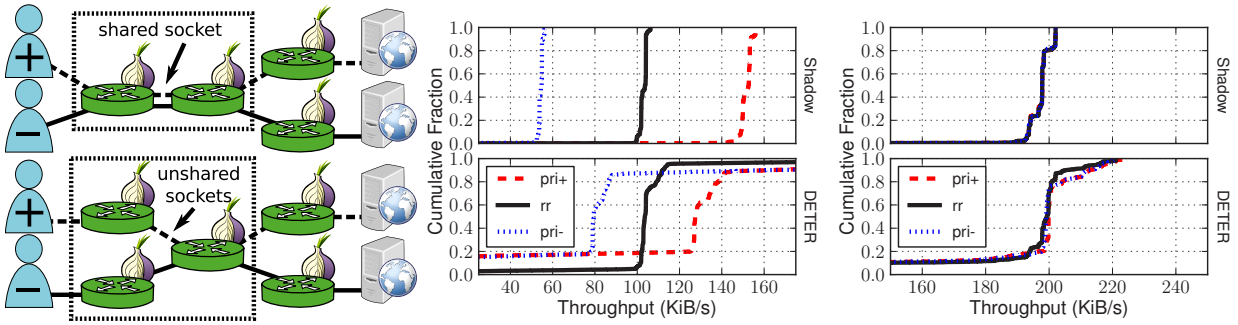
5.1 Mismanaged Socket Output

As described in Section 2, each Tor relay creates and maintains a single TCP connection to every relay to which it is connected. All communication between two relays occurs through this single TCP connection channel. In particular, this channel multiplexes all circuits that are established between its relay endpoints. TCP provides Tor a reliable and in-order data transport.

Sequential Socket Writes: Tor uses the asynchronous event library `libevent` [4] to assist with sending and receiving data to and from the kernel (i.e. network). Each TCP connection is represented as a socket in the kernel, and is identified by a unique socket descriptor. Tor registers each socket descriptor with libevent, which itself manages kernel polling and triggers an asynchronous notification to Tor via a callback function of the readability and writability of that socket. When Tor receives this notification, it chooses to read or write as appropriate.

An important aspect of these libevent notifications is that they happen for *one socket at a time*, regardless of the number of socket descriptors that Tor has registered. Tor attempts to send or receive data from that one socket without considering the state of any of the other sockets. This is particularly troublesome when writing, as Tor will only be able to choose from the non-empty circuits belonging to the currently triggered socket and no other. Therefore, Tor's circuit scheduler may schedule a circuit with worse priority than it would have if it could choose from all sockets that are able to be triggered at that time. Since the kernel schedules with a first-come first-serve (FCFS) discipline, Tor may actually be sending data out of priority order simply due to the order in which the socket notifications are delivered by libevent.

Bloated Socket Buffers: Linux uses TCP autotuning to dynamically and monotonically increase each



(a) Socket Sharing Scenarios (b) Throughput with Shared Socket (c) Throughput with Unshared Sockets

Figure 5: Socket sharing affects circuit priority scheduling.

socket buffer's capacity using the socket connection's bandwidth-delay product calculation [61]. TCP auto-tuning increases the amount of data the kernel will accept from the application in order to ensure that the socket is able to fully utilize the network link. TCP auto-tuning is an extremely useful technique to maximize throughput for applications with few sockets or without priority requirements. However, it may cause problems for more complex applications like Tor.

When libevent notifies Tor that a socket is writable, Tor writes as much data as possible to that socket (i.e., until the kernel returns `EWOULDBLOCK`). Although this improves utilization when only a few auto-tuned sockets are in use, consider a Tor relay that writes to thousands of auto-tuned sockets (a common situation since Tor maintains a socket for every relay with which it communicates). These sockets will *each* attempt to accept enough data to fully utilize the link. If Tor fills all of these sockets to capacity, the kernel will clearly be unable to immediately send it all to the network. Therefore, with many active sockets in general and for asymmetric connections in particular, the potential for kernel queuing delays are dramatic. As we have shown in Section 4, writing as much as possible to the kernel as Tor currently does results in large kernel queuing delays.

Tor can no longer adjust data priority once it is sent to the kernel, even if that data is still queued in the kernel when Tor receives data of higher importance later. To demonstrate how this may result in poor scheduling decisions, consider a relay with two circuits: one contains sustained, high throughput traffic of worse priority (typical of many bulk data transfers), while the other contains bursty, low throughput traffic of better priority (typical of many interactive data transfer sessions). In the absence of data on the low throughput circuit, the high throughput circuit will fill the entire kernel socket buffer whether or not the kernel is able to immediately send that data. Then, when a better priority cell arrives, Tor will immediately schedule and write it to the kernel. However, since the kernel sends data to the network in the same order in which it was received from the application (FCFS), that better priority cell data must wait until all of the previously received high throughput data is flushed to the network. This problem theoretically worsens as the number of sockets increase, suggesting that recent research proposing that Tor use multiple sockets between each pair of relays [16, 26] may be misguided.

Effects on Circuit Priority: To study the effects on circuit priority, we customized Tor as follows. First, we added the ability for each client to send a special cell after building a circuit that communicates one of two priority classes to the circuit's relays: a better priority class; or a worse priority class. Second, we customized the built-in EWMA circuit scheduler that prioritizes bursty traffic over bulk traffic [57] to include a priority factor $\mathscr{F}$: the circuit scheduler counts $\mathscr{F}$ cells for every cell scheduled on a worse priority circuit. Therefore, the EWMA of the worse priority class will effectively increase $\mathscr{F}$ times faster than normal, giving a scheduling advantage to better priority traffic.

We experiment with two separate private Tor networks: one using Shadow [35], a discrete event network simulator that runs Tor in virtual processes; and the other using DETER [3], a custom experimentation testbed that runs Tor on bare-metal hardware. We consider two clients downloading from two file servers through Tor in the scenarios shown in Figure 5a:

- *shared socket:* the clients share entry and middle relays, but use different exit relays – the clients' circuits each belong to the same socket connecting the middle to the entry; and
- *unshared sockets:* the clients share only the middle relay – the clients' circuits each belong to independent sockets connecting the middle to each entry.

We assigned one client's traffic to the better priority class (denoted with "+") and the other client's traffic to the worse priority class (denoted with "-"). We configured all nodes with a 10 Mbit symmetric access link, and approximated a middle relay bottleneck by setting its socket buffer size to 32 KiB. Our configuration allows us

to focus on the socket contention that will occur at the middle relay, and the four cases that result when considering whether or not the two circuits share incoming or outgoing TCP connections at the middle relay. Clients downloaded data through the circuits continuously for 10 minutes in Shadow and 60 minutes on DETER.[6]

The results collected during each of the scenarios are shown in Figure 5. Plotted is the cumulative distribution of the throughput achieved by the better ("pri+") and worse ("pri-") priority clients using the priority scheduler, as well as the combined cumulative distribution for both clients using Tor's default round-robin scheduler ("rr"). As shown in Figure 5b, performance differentiation occurs correctly with the priority scheduler on a shared socket. However, as shown in Figure 5c, the priority scheduler is unable to differentiate throughput when the circuits do not share a socket.

Discussion: As outlined above, the reason for no differentiation in the case of the unshared socket is that both circuits are treated independently by the scheduler due to the sequential libevent notifications and the fact that Tor currently schedules circuits belonging to one socket at a time while ignoring the others. We used TorPS [10], a Tor path selection simulator, to determine how often we would expect unshared sockets to occur in practice. We used TorPS to build 10 million paths following Tor's path selection algorithm, and computed the probability of two circuit paths belonging to each scenario. We found that any two paths may be classified as unshared (they share at least one relay but never share an outgoing socket) at least 99.775 percent of the time, clearly indicating that adjusting Tor's socket management may have a dramatic effect on data priority inside of Tor.

Note that the socket mismanagement problem is not solved simply by parallelizing the libevent notification system and priority scheduling processes (which would require complex code), or by utilizing classful queuing disciplines in the kernel (which would require root privileges); while these may improve control over traffic priority to some extent, they would still result in bloated buffers containing data that cannot be sent due to closed TCP congestion windows.

5.2 The KIST Algorithm

In order to overcome the inefficiencies resulting from Tor's socket management, KIST chooses between *all circuits* that have queued data *irrespective* of the socket to which the circuit belongs, and *dynamically adjusts the amount written* to each socket based on real-time kernel information. We now detail each of these approaches.

[6]The small-scale experiments described here are meant to isolate Tor's internal queuing behavior for analysis purposes, and do not fully represent the live Tor network, its background traffic, or its load.

Algorithm 1 The KIST `NotifySocketWritable()` callback, invoked by libevent for each writable socket.

Require: $sdesc, conn, \mathscr{T} \leftarrow GlobalWriteTimeout$

1: $L_p \leftarrow getPendingConnectionList()$
2: **if** L_p **is** $Null$ **then**
3: $\quad L_p \leftarrow newList()$
4: $\quad setPendingConnectionList(L_p)$
5: $\quad createCallback(\mathscr{T}, \texttt{NotifyGlobalWrite()})$
6: **end if**
7: **if** $L_p.contains(conn)$ **is** $False$ **then**
8: $\quad L_p.add(conn)$
9: **end if**
10: $disableNotify(sdesc)$

Global Circuit Scheduling: Recall that libevent delivers write notification events for a single socket at a time. Our approach with KIST is relatively straightforward: rather than handle the kernel write task immediately when libevent notifies Tor that a socket is writable, we simply collect a set of sockets that are writable over a time interval specified by an adjustable `GlobalWriteTimeout` parameter. This allows us to increase the number of candidate circuits we consider when scheduling and writing cells to the kernel: we may select among all circuits which contain cells that are waiting to be written to one of the sockets in our writable set.

The socket collection approach is outlined in Algorithm 1. The socket descriptor `sdesc` and a connection state object `conn` are supplied by libevent. Note that we disable notification events for the socket (as shown in line 10) in order to prevent duplicate notification events during the socket collection interval.

After the `GlobalWriteTimeout` time interval, KIST begins writing cells to the sockets according to the circuit scheduling policy. There are two major phases to this process, which is outlined in Algorithm 2. In lines 4 and 8, we distinguish sockets that contain raw bytes ready to be written directly to the kernel (previously scheduled cells with TLS headers attached) from those with additional cells ready to be converted to raw bytes. KIST first writes the already scheduled raw bytes (lines 4-7), and then schedules and writes additional cells after converting them to raw bytes and adding TLS headers (lines 13-15). Note that the connections should be enumerated (on line 3 of Algorithm 2) in an order that respects the order in which cells were converted to raw bytes by the circuit scheduler in the previous round.

The global scheduling approach does not by itself solve the bloated socket buffer problem. KIST also dynamically computes socket write limits on line 2 of Algorithm 2 using real-time TCP, socket, and bandwidth information, which it then uses when deciding how much to write to the kernel.

Algorithm 2 The KIST `NotifyGlobalWrite()` callback, invoked after the `GlobalWriteTimeout` period.

1: $L_{eligible} \leftarrow newList()$
2: $K \leftarrow collectKernelInfo(getConnectionList())$
3: **for all** *conn* **in** $getPendingConnectionList()$ **do**
4: **if** $hasBytesForKernel(conn)$ **is** $True$ **then**
5: $enableNotify(conn)$
6: $nBytes \leftarrow writeBytesToKernel(K, conn)$
7: **end if**
8: **if** $hasCells(conn)$ **is** $True$ **and** $getLimit(K, conn) > 0$ **then**
9: $L_{eligible}.add(conn)$
10: **end if**
11: **end for**
12: **while** $L_{eligible}.isEmpty()$ **is** $False$ **do**
13: $conn \leftarrow scheduleCell(L_{eligible})$ {cell to bytes}
14: $enableNotify(conn)$
15: $nBytes \leftarrow writeBytesToKernel(K, conn)$
16: **if** $nBytes$ **is** 0 **or** $getLimit(K, conn)$ **is** 0 **then**
17: $L_{eligible}.remove(conn)$
18: **end if**
19: **end while**

Managing Socket Output: KIST attempts to move the queuing delays from the kernel outbound queue to Tor's circuit queue by keeping kernel output buffers as small as possible, i.e., by only writing to the kernel as much as the kernel will actually send. By delaying the circuit scheduling decision until the last possible instant before kernel starvation occurs, Tor will ultimately improve its control over the priority of outgoing data. This approach attempts to give Tor approximately the same control over outbound data that it would have if it had direct access to the network interface. When combined with global circuit scheduling, Tor's influence over outgoing data priority should improve.

To compute write limits, KIST first makes three system calls for each connection: `getsockopt` on level `SOL_SOCKET` for option `SO_SNDBUF` to get *sndbufcap*, the capacity of the send buffer; `ioctl` with command `SIOCOUTQ` to get *sndbuflen*, the current length of the send buffer; and `getsockopt` on level `SOL_TCP` for option `TCP_INFO` to get *tcpi*, a variety of TCP state information. The TCP information used by KIST includes the connection's maximum segment size *mss*, the congestion window *cwnd*, and the number of *unacked* packets for which the kernel is waiting for an acknowledgment from the TCP peer. KIST then computes a write limit for each connection c as follows:

$$\begin{aligned} socket\_space_c &= sndbufcap_c - sndbuflen_c \\ tcp\_space_c &= (cwnd_c - unacked_c) \cdot mss_c \\ limit_c &= \min(socket\_space_c, tcp\_space_c) \end{aligned} \quad (1)$$

The key insight in Equation 1 is that TCP will not allow the kernel to send more packets than dictated by the congestion window, and that the unacknowledged packets prevent the congestion window from sliding open. By respecting this write limit for each connection, KIST ensures that the data sent to the kernel is immediately sendable and reduces kernel queuing delays.

If all connections are sending data in parallel, it is still possible to overwhelm the kernel with more data than it can physically send to the network. Therefore, KIST also computes a global write limit at the beginning of each `GlobalWriteTimeout` period:

$$\begin{aligned} sndbuflen\_prev &= sndbuflen \\ sndbuflen &= \textstyle\sum_{c_i} \left(sndbuflen_{c_i}\right) \\ bytes\_sent &= sndbuflen - sndbuflen\_prev \\ limit &= \max(limit, bytes\_sent) \end{aligned} \quad (2)$$

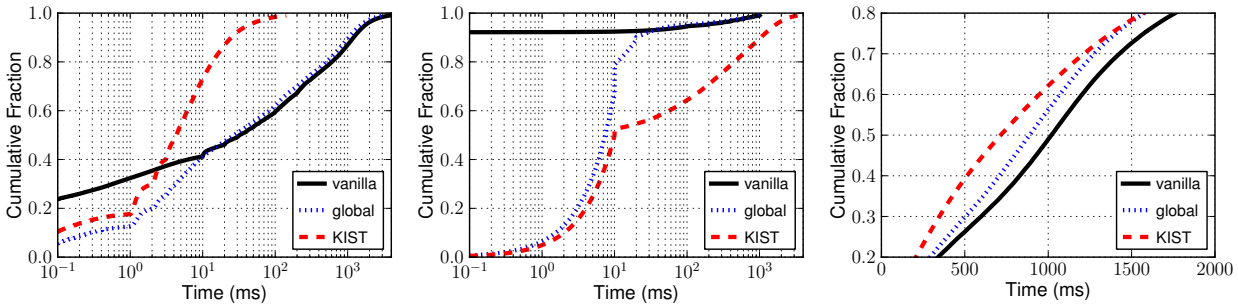
Note that Equation 2 is an attempt to measure the actual upstream bandwidth speed of the machine. In practice, this could be done in a testing phase during which writes are not limited, configured manually, or estimated using other techniques such as packet trains [32].

The connection and global limits are computed at the beginning of a scheduling round, i.e., on line 2 of Algorithm 2; they are enforced whenever bytes are written to the kernel, i.e., on lines 6 and 15 of Algorithm 2. Note that they will be bounded above by Tor's independently configured connection and global application rate limits.

5.3 Experiments and Results

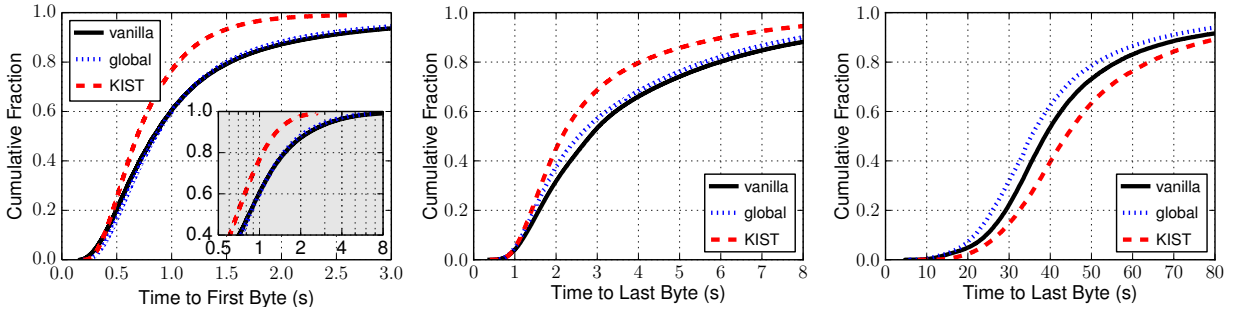
We use Shadow and its models as discussed in Section 3 to measure KIST's effect on network performance, congestion, and throughput. We also evaluate its CPU overhead. See Appendix C [34] for an analysis under a more heavily loaded Shadow-Tor network. Note that we found that KIST performs as well or better under heavier load than under normal load as presented in this section, indicating that it can gracefully scale as Tor grows.

Prototype: We implemented a KIST protoype as a patch to Tor version `0.2.3.25`, and included the elements discussed in Section 4 necessary for measuring congestion during our experiments. We tested vanilla Tor using the default `CircuitPriorityHalflife` of 30, the global scheduling part of KIST (without enforcing the write limits), and the complete KIST algorithm. We configured the global scheduler to use a 10 millisecond `GlobalWriteTimeout` in both the global and KIST experiments. Note that our KIST implementation ignores the connection enumeration order on line 3 of Algorithm 2, an optimization that may further improve Tor's control over priority in cases where the global limit is reached before the algorithm reaches line 12.



(a) Kernel Out Congestion (b) Tor Congestion (c) Circuit Congestion

Figure 6: Congestion for vanilla Tor, KIST, and the global scheduling part of KIST (without enforcing write limits). Figures 6a and 6b show the distribution of cell congestion local to each relay (with logarithmic x-axes), while Figure 6c shows the distribution of the end-to-end circuit congestion for all measured cells.



(a) All Clients (b) 320 KiB "web" clients (c) 5 MiB "bulk" clients

Figure 7: Client performance for vanilla Tor, KIST, and the global scheduling part of KIST (without enforcing write limits). Figure 7a shows the distribution of the time until the client receives the first byte of the data payload, for all clients, while the inset graph shows the same distribution with a logarithmic x-axis. Figures 7b and 7c show the distribution of time to complete a 320 KiB and 5 MiB file by the "web" and "bulk" clients, respectively.

Congestion: Recall that the goal of KIST is to move congestion from the kernel outbound queue to Tor where it can better be managed. Figure 6 shows KIST's effectiveness in this regard. In particular, Figure 6a shows that KIST reduces kernel outbound congestion over vanilla Tor by one to two order of magnitude for over 40 percent of the sampled cells. Further, it shows that the queue time is less than 200 milliseconds for 99 percent of the cells measured, compared to over 4000 milliseconds for both vanilla Tor and global scheduling alone.

Figure 6b shows how global scheduling and KIST increase the congestion inside of Tor. Both global scheduling and KIST result in sharp Tor queue time increases up to 10 milliseconds, after which the existing 10 millisecond `GlobalWriteTimeout` timer event will fire and Tor will flush more data to the kernel. With global scheduling, most of the data queued in Tor quickly gets transferred to the kernel following this timeout, whereas data is queued inside of Tor much longer when using KIST. This result is an explicit feature of KIST, as it means Tor will have more control over data priority when scheduling circuits.

While we have shown above how KIST is able to move congestion from the kernel into Tor, Figure 6c shows the aggregate effect on cell congestion during its complete existence through the entire end-to-end circuit. KIST reduces aggregate circuit congestion from 1010.1 milliseconds to 704.5 milliseconds in the median, a 30.3 percent improvement, while global scheduling reduces congestion by 13 percent to 878.8 milliseconds.

The results in Figure 6 show that KIST indeed achieves its congestion management goals while highlighting the importance of limiting kernel write amounts in addition to globally scheduling circuits.

Performance: We show in Figure 7 how KIST affects client performance. Figure 7a shows how network latency is generally affected by showing the time until the first byte of every download by all clients. Global scheduling alone is roughly indistinguishable from vanilla Tor, while KIST reduces latency to the first byte for over 80 percent of the downloads—in the median, KIST reduces network latency by 18.1 percent from 0.838 seconds to 0.686 seconds. The inset graph has a logarithmic x-axis and shows that KIST is particularly

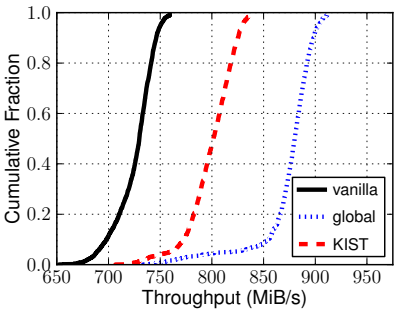


Figure 8: Aggregate relay write throughput for vanilla Tor, KIST, and the global scheduling part of KIST (without enforcing write limits). Both of our enhancements increase network throughput over vanilla Tor.

beneficial in the upper parts of the distribution: in the 99th percentile, latency is reduced from more than 7 seconds to less than 2.7 seconds.

Figures 7b and 7c show the distribution of time to complete each 320 KiB download for the "web" clients and each 5 MiB file for the "bulk" clients, respectively. In our experiments, the 320 KiB download times decreased by over 1 second for over 40 percent of the downloads, while the download times for 5 MiB files increased by less than 8 seconds for all downloads. These changes in download times are a result of Tor correctly utilizing its circuit priority scheduler, which prioritizes traffic with the lowest exponentially-weighted moving average throughput. As the "web" clients pause between downloads, their traffic is often prioritized ahead of "bulk" traffic. Our results indicate that not only does KIST decrease Tor network latency, it also increases Tor's ability to appropriately manage its traffic.

Throughput: We show in Figure 8 KIST's effect on relay throughput. Shown is the distribution of aggregate bytes written per second by all relays in the network. We found that throughput improves when using KIST due to a combination of the reduction in network latency and our client model: web clients completed their downloads faster in the lower latency network and therefore also downloaded more files. By lowering circuit congestion, KIST improves utilization of existing bandwidth resources over vanilla Tor by 71.6 MiB/s, or 9.8%, in the median. While the best network utilization is achieved with global scheduling without write limits (a 150.1 MiB/s, or 20.5%, improvement over vanilla Tor in the median), we have shown above that it is less effective than KIST at reducing kernel congestion and allowing Tor to correctly prioritize traffic.

Overhead: The main overhead in KIST involves the collection of socket and TCP information from the kernel using three separate calls to `getsockopt` (socket capacity, socket length, and TCP info). These three system calls are made for every connection after every `GlobalWriteTimeout` interval. To understand the overhead involved with these calls, we instrumented curiosity3 from Section 4 to collect timing information while performing the syscalls required by KIST. Our test ran on the live relay running an Intel Xeon x3450 CPU at 2.67GHz for 3 days and 14 hours, and collected a timing sample every second for a total of 309,739 samples. We found that the three system calls took 0.9140 microseconds per connection in the median, with a mean of 0.9204 and a standard deviation of 3.1×10^{-5}.

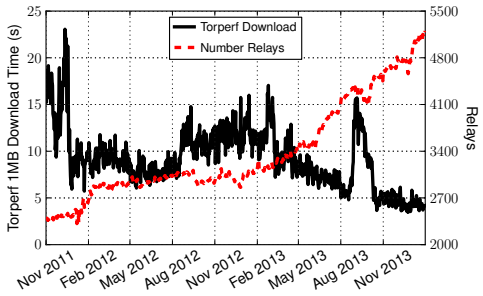


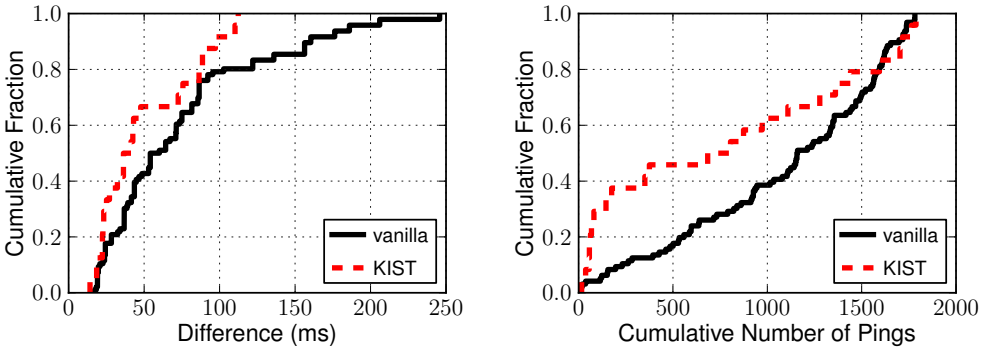
Figure 9: Network size correlates with performance.

The number of connections a relay may have is bounded above roughly by the number of relays in the network, which is currently around 5,000. Therefore, we expect the overhead to be less than 5 milliseconds and reasonable for current relays. If this overhead becomes problematic as Tor grows, the gathering of kernel information can be outsourced to a helper thread and continuously updated over time. Further, we have determined through discussions with Linux kernel developers that the *netlink socket diag* interface could be used to collect information for several sockets at once—an optimization that may provide significant reductions in overhead.

6 Security Analysis

Performance and Security: Performance and ease of use affect adoption rates of any network technology. They have played a central role in the size and diversity of the Tor userbase. This can then affect the size of the network itself as users are more willing to run parts of the network or contribute financially to its upkeep, e.g., via torservers.net. Growth from performance improvements affect the security of Tor by increasing the uncertainty for many types of adversaries concerning who is communicating with whom [17, 20, 22, 37]. Performance factors in anonymous communication systems like Tor are thus pertinent to security in a much more direct way than they typically would be for, say, a faster signature algorithm's impact on the security of an authentication system.

Though real and more significant, direct effects of performance on Tor's security from network and userbase growth are also hard to show, given both the variety of



(a) Latency Estimate Difference

(b) Cumulative Pings until Best Estimate

Figure 10: Latency leaks are more pronounced (10a) and are faster (10b) with KIST.

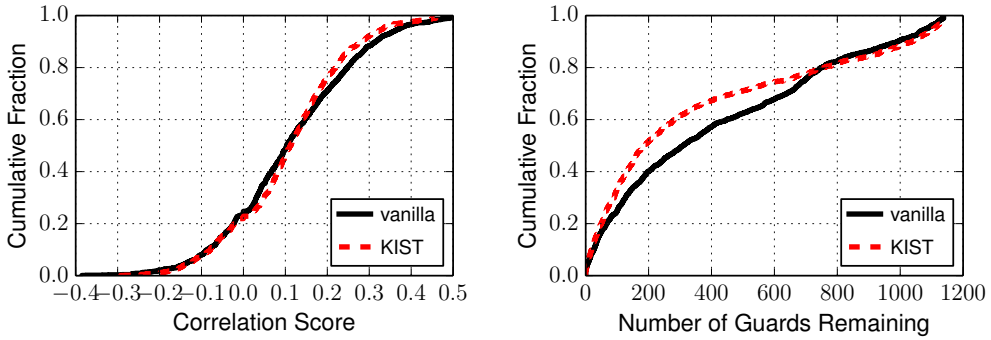
causal factors and the difficulty of gathering useful data while preserving privacy. Whatever the causal explanation, a correlation ($-.62$) between Tor performance improvement over time (measured by the median download time of a 1 MB file) and network size is shown in Figure 9. (Numbers are from the Tor Metrics Portal [8].) Similar results hold when number of relays is replaced with bandwidth metrics. Which is more relevant depends on the adversary's most significant constraints: adversary size and distribution across the underlying network are important considerations [39].

More measurable effects can occur if a performance change creates a new opportunity for attack or makes an existing attack more effective or easier to mount. Performance change may also eliminate or diminish previous possible or actual attacks. Growth effects are potentially the greatest security effects of our performance changes, but we now focus on these more directly observable aspects. They include attacks on Tor based on resource contention or interference [19, 24, 25, 31, 44, 46, 48] or simply available resource observation [44], or observing other performance properties, such as latency [31]. Many papers have also explored improvements to Tor performance via scheduling, throttling, congestion management, etc. (see Section 2). Manipulating performance enhancement mechanisms can turn them into potential vectors of attack themselves [38]. Geddes *et al.* [25] analyzed anonymity impact of several performance enhancement mechanisms for Tor.

Latency Leak: The basic idea of a latency leak attack as first set out in [30] is to measure RTT (roundtrip time) between a compromised exit and the client of some target connection repeatedly and then to pick the shortest result as an indication of latency. Next compare this to the known, measured latency through all the hops in the circuit except client to entry relay. (Other attacks such as throughput measurement, discussed below, are assumed to have already identified the relays in the circuit.) Next, use that to determine the latency between the client of the target connection and its entry relay, which is in a known network location. This can significantly reduce the range of possible network locations for the client. When measuring latency using our improved models and simulator, we discovered that this attack is generally able to determine latency well with vanilla Tor. While KIST improves the overall accuracy, the improvement is small when a good estimate was also found with vanilla Tor.

Figure 10a shows the results of an experiment run on our model from Section 3 with random circuit and client choices, indicating the difference between the correct latency and the estimate after a few hundred pings once per second. Roughly 20% of circuits for both vanilla Tor and KIST are within 25ms of the correct latency. After this they diverge, but both have a median latency estimate of about 50ms or less. It is only for the worst 10-20% of estimates, which are presumably not useful anyway, that KIST is substantially better. While the eventual accuracy of the attack is comparable for both, the attacker under KIST is significantly faster on average. Figure 10b shows the cumulative number of pings (seconds) until a best estimate is achieved. After 200 pings, nearly 40% of KIST circuits have achieved their best estimate while less than 10% have for vanilla Tor. And the median number of pings needed for KIST is about 700 vs. 1200 for vanilla Tor.

The accuracy of the latency attack indicated above is a significant threat to network location, which from a technical perspective is what Tor is primarily designed to protect. It could be diminished by padding latency. Specifically any connection at the edges of the Tor network, at either source or destination end, could be dynamically padded by the entry or exit relay respectively to ideally make latency of all edge connections through that relay uniform—more realistically to significantly decrease the network location information leaked by latency. Relays can do their own RTT measurements for any edge



(a) Throughput Correlation, Probe to Target Client (b) Guards Remaining in Candidate Guard Set

Figure 11: While the aggregate throughput correlations of the probe to the true guard over the set of all probes (11a) are not significantly affected by KIST, it is slightly easier for the adversary to eliminate candidate guards of a target client (for all clients) (11b) when using KIST.

connection and pad accordingly. There are many issues around this suggestion, which we leave to future work.

Throughput Leak: The throughput attack introduced by Mittal *et al.* [44] identifies the entry relay of a target connection by setting up one-hop probe circuits from attack clients through all prospective entry relays and back to the client in order to measure throughput at those relays. These throughputs are compared to the throughput observed by the exit relay of the target circuit. The attack is directed against circuits used for bulk downloading since these will attempt a sustained maximum throughput, and will result in congestion effects on bottleneck relays that allow the adversary to reduce uncertainty about possible entry relays. Mittal *et al.* also looked at attacks on lower bandwidth interactive traffic and found some success, although with much less accuracy than for bulk traffic.

We analyze the extent to which KIST affects the throughput attack. While measuring throughput at entry relays, we also adopt the simplification of Geddes *et al.* [25] of restricting observations to entry relays that are not used as middle or exit relays for other bulk downloading circuits. This allows us the efficiency of making measurements for several simulated attacks simultaneously while minimizing interference between their probes.

Figure 11a shows the cumulative distribution of scores for correlation of probe throughput at the correct entry relay with throughput at the observed exit relay under vanilla Tor and under KIST scheduling (on the network and user model given in Section 3). Throughput was measured every 100 ms. We found that the throughput correlations are not significantly affected by KIST.

To explain the correlation scores, recall from Section 5.2 how KIST reduces both circuit congestion and network latency by allowing Tor to properly prioritize circuits independent of the TCP connections to which they belong. This leads to two competing potential effects on the throughput attack: (1) a less congested network will increase the sensitivity of the probes to variations in throughput, thereby allowing stronger correlations between the throughput achieved by the probe client and that achieved by the target client; and (2) a circuit's throughput is most correlated with that of its bottleneck relay, and KIST's improved scheduling should also reduce the bottleneck effects of congestion in the network and allow weaker throughput correlations. Further, the improved priority scheduling (moving from round-robin over TCP connections to properly utilizing EWMA over circuits) will cause the throughput of each client to become slightly "burstier" over the short term as the priority causes the scheduler to oscillate between the circuits. We suspect that the similar correlation scores are the result of combining these effects.

To further understand KIST's affect on the throughput attack, we measure how the correlation of every client's throughput to the *true* guard's throughput compares to the correlation of the client's throughput to that of every other *candidate* guard in the network. For every client, we start with a candidate guard set of all guards, and remove those guards with a lower correlation score with the client than the true guard's score. Figure 11b shows the distribution, over all clients, of the extent to which we were able to reduce the size of the candidate guard set using this heuristic. Although KIST reduced the uncertainty about the true guard used by the target client, we do not expect the small improvement to significantly affect the ability to conduct a successful throughput attack in practice.

7 Conclusion

In this paper, we outlined the results of an in-depth congestion study using both public and private Tor net-

works. We identified that most congestion occurs in outbound kernel buffers, analyzed Tor socket management, and designed a new socket transport mechanism called KIST. Through evaluation in a full-scale private Shadow-Tor network, we conclude that KIST is capable of moving congestion into Tor where it can be better managed by application priority scheduling mechanisms. More specifically, we found that by considering all sockets and respecting TCP state information when writing data to the kernel, KIST reduces both congestion and latency while increasing utilization. Finally, we performed a detailed evaluation of KIST against well-known latency and throughput attacks. While KIST increases the speed at which true network latency can be calculated, it does not significantly affect the accuracy of the probes required to correlate throughput.

Future work should extend our simulation-based evaluation and consider how KIST performs for relays in the live Tor network. We note that our analysis is based exclusively on Linux relays, as 91% of Tor's bandwidth is provided by relays running a Linux-based distribution [58]. Although we expect KIST to improve performance similarly across platforms because it primarily works by managing socket buffer levels, future work should consider how KIST is affected by the interoperation of relays running on a diverse set of OSes. Finally, our KIST prototype would benefit from optimizations, particularly by running the process of gathering kernel state information in a separate thread and/or using the *netlink socket diag* interface.

Acknowledgments

We thank our shepherd, Rachel Greenstadt, and the anonymous reviewers for providing feedback that helped improve this work. We thank Roger Dingledine for discussions about measuring congestion in Tor, and Patrick McHardy for suggesting the use of the netlink socket diag interface. This work was partially supported by ONR, DARPA, and the National Science Foundation through grants CNS-1149832, CNS-1064986, CNS-1204347, CNS-1223825, and CNS-1314637. This material is based upon work supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency and Space and Naval Warfare Systems Center Pacific.

References

[1] Alexa top 1 million sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip. Retrieved 2012-01-31.

[2] CAIDA data. http://www.caida.org/data.

[3] DETER testbed. http://www.isi.edu/deter.

[4] libevent event notification library. http://libevent.org/.

[5] libkqtime code repository. https://github.com/robgjansen/libkqtime.git.

[6] libpcap portable C/C++ library for network traffic capture. http://www.tcpdump.org/.

[7] Shadow homepage and code repositories. https://shadow.github.io/, https://github.com/shadow/.

[8] Tor Metrics Portal. http://metrics.torproject.org/.

[9] TorPerf. https://gitweb.torproject.org/torperf.git/.

[10] TorPS homepage. http://torps.github.io/.

[11] AKHOONDI, M., YU, C., AND MADHYASTHA, H. V. LASTor: A low-latency as-aware Tor client. In *IEEE Symposium on Security and Privacy (Oakland)* (2012).

[12] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[13] ALSABAH, M., BAUER, K., ELAHI, T., AND GOLDBERG, I. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *Privacy Enhancing Technologies Symposium (PETS)* (2013).

[14] ALSABAH, M., BAUER, K., AND GOLDBERG, I. Enhancing Tor's performance using real-time traffic classification. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[15] ALSABAH, M., BAUER, K., GOLDBERG, I., GRUNWALD, D., MCCOY, D., SAVAGE, S., AND VOELKER, G. DefenestraTor: Throwing out windows in Tor. In *Privacy Enhancing Technologies Symposium (PETS)* (2011).

[16] ALSABAH, M., AND GOLDBERG, I. PCTCP: Per-circuit tcp-over-ipsec transport for anonymous communication overlay networks. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[17] BACK, A., MÖLLER, U., AND STIGLIC, A. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Workshop on Information Hiding (IH)* (2001).

[18] CHAABANE, A., MANILS, P., AND KAAFAR, M. Digging into anonymous traffic: A deep analysis of the tor anonymizing network. In *IEEE Conference on Network and System Security (NSS)* (2010).

[19] CHAN-TIN, E., SHIN, J., AND YU, J. Revisiting circuit clogging attacks on Tor. In *IEEE Conference on Availability, Reliability and Security (ARES)* (2013).

[20] DINGLEDINE, R., AND MATHEWSON, N. Anonymity loves company: Usability and the network effect. In *Workshop on the Economics of Information Security (WEIS)* (2006).

[21] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *USENIX Security Symposium (USENIX)* (2004).

[22] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Deploying low-latency anonymity: Design challenges and social factors. *IEEE Security & Privacy 5*, 5 (Sept./Oct. 2007), 83–87.

[23] DINGLEDINE, R., AND MURDOCH, S. J. Performance improvements on Tor or, why Tor is slow and what we're going to do about it. Tech. Rep. 2009-11-001, The Tor Project, 2009.

[24] EVANS, N. S., DINGLEDINE, R., AND GROTHOFF, C. A practical congestion attack on Tor using long paths. In *USENIX Security Symposium (USENIX)* (2009).

[25] GEDDES, J., JANSEN, R., AND HOPPER, N. How low can you go: Balancing performance with anonymity in Tor. In *Privacy Enhancing Technologies Symposium (PETS)* (2013).

[26] GOPAL, D., AND HENINGER, N. Torchestra: Reducing interactive traffic delays over Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)* (2012).

[27] HA, S., RHEE, I., AND XU, L. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 64–74.

[28] HAHN, S., AND LOESING, K. Privacy-preserving ways to estimate the number of Tor users. Tech. Rep. 2010-11-001, The Tor Project, 2010.

[29] HOPPER, N. Protecting Tor from botnet abuse in the long term. Tech. Rep. 2013-11-001, The Tor Project, 2013.

[30] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? In *ACM Conference on Computer and Communications Security (CCS)* (2007). Expanded and revised version in [31].

[31] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC) 13*, 2 (Feb. 2010), 13–28.

[32] JAIN, R., AND ROUTHIER, S. Packet trains–measurements and a new model for computer network traffic. *IEEE Selected Areas in Communications 4*, 6 (1986), 986–995.

[33] JANSEN, R., BAUER, K., HOPPER, N., AND DINGLEDINE, R. Methodically modeling the Tor network. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)* (2012).

[34] JANSEN, R., GEDDES, J., WACEK, C., SHERR, M., AND SYVERSON, P. Appendices to accompany "Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport". Tech. Rep. 14-012, Univ. of Minnesota, 2014. http://www.cs.umn.edu/tech_reports_upload/tr2014/14-012.pdf.

[35] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a box for accurate and efficient experimentation. In *USENIX Security Symposium (USENIX)* (2012).

[36] JANSEN, R., HOPPER, N., AND KIM, Y. Recruiting new Tor relays with BRAIDS. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[37] JANSEN, R., JOHNSON, A., AND SYVERSON, P. LIRA: Lightweight incentivized routing for anonymity. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[38] JANSEN, R., SYVERSON, P., AND HOPPER, N. Throttling Tor bandwidth parasites. In *USENIX Security Symposium (USENIX)* (2012).

[39] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users get routed: Traffic correlation on tor by realistic adversaries. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[40] MATHEWSON, N. Evaluating SCTP for Tor. http://archives.seul.org/or/dev/Sep-2004/msg00002.html, Sept. 2004. Listserv posting.

[41] MATHIS, M., AND MAHDAVI, J. Forward acknowledgement: Refining TCP congestion control. *ACM SIGCOMM Computer Communication Review 26*, 4 (1996), 281–291.

[42] MATHIS, M., MAHDAVI, J., FLOYD, S., AND ROMANOW, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

[43] MCCOY, D., BAUER, K., GRUNWALD, D., KOHNO, T., AND SICKER, D. Shining light in dark places: Understanding the Tor network. In *Privacy Enhancing Technologies Symposium (PETS)* (2008).

[44] MITTAL, P., KHURSHID, A., JUEN, J., CAESAR, M., AND BORISOV, N. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *ACM Conference on Computer and Communications Security (CCS)* (2011).

[45] MOORE, W. B., WACEK, C., AND SHERR, M. Exploring the potential benefits of expanded rate limiting in Tor: Slow and steady wins the race with Tortoise. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[46] MURDOCH, S. J. Hot or not: Revealing hidden services by their clock skew. In *ACM Conference on Computer and Communications Security (CCS)* (2006).

[47] MURDOCH, S. J. Comparison of Tor datagram designs. Tech. Rep. 2011-11-001, The Tor Project, 2011.

[48] MURDOCH, S. J., AND DANEZIS, G. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy (Oakland)* (2005).

[49] NOWLAN, M. F., TIWARI, N., IYENGAR, J., AMIN, S. O., AND FORD, B. Fitting square pegs through round pipes. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[50] NOWLAN, M. F., WOLINSKY, D., AND FORD, B. Reducing latency in Tor circuits with unordered delivery. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)* (2013).

[51] The ns2 Network Simulator. http://www.isi.edu/nsnam/ns/.

[52] PAXSON, V., ALLMAN, M., CHU, J., AND SARGENT, M. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[53] RAMACHANDRAN, S. Web metrics: Size and number of resources. https://developers.google.com/speed/articles/web-metrics, 2010.

[54] REARDON, J., AND GOLDBERG, I. Improving Tor using a TCP-over-DTLS tunnel. In *USENIX Security Symposium (USENIX)* (2009).

[55] SHERR, M., MAO, A., MARCZAK, W. R., ZHOU, W., LOO, B. T., AND BLAZE, M. A3: An Extensible Platform for Application-Aware Anonymity. In *Network and Distributed System Security Symposium (NDSS)* (2010).

[56] SNADER, R., AND BORISOV, N. A tune-up for Tor: Improving security and performance in the Tor network. In *Network and Distributed System Security Symposium (NDSS)* (2008).

[57] TANG, C., AND GOLDBERG, I. An improved algorithm for Tor circuit scheduling. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[58] Tor network status. http://torstatus.blutmagie.de/index.php.

[59] WACEK, C., TAN, H., BAUER, K., AND SHERR, M. An empirical evaluation of relay selection in Tor. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[60] WANG, T., BAUER, K., FORERO, C., AND GOLDBERG, I. Congestion-aware path selection for Tor. In *Financial Cryptography and Data Security (FC)* (2012).

[61] WEIGLE, E., AND FENG, W.-C. A comparison of TCP automatic tuning techniques for distributed computing. In *IEEE Symposium on High Performance Distributed Computing (HPDC)* (2002).

Effective Attacks and Provable Defenses for Website Fingerprinting

Tao Wang[1] Xiang Cai[2] Rishab Nithyanand[2] Rob Johnson[2] Ian Goldberg[1]

[1]University of Waterloo
{t55wang,iang}@cs.uwaterloo.ca

[2]Stony Brook University
{xcai,rnithyanand,rob}@cs.stonybrook.edu

Abstract

Website fingerprinting attacks allow a local, passive eavesdropper to identify a user's web activity by leveraging packet sequence information. These attacks break the privacy expected by users of privacy technologies, including low-latency anonymity networks such as Tor. In this paper, we show a new attack that achieves significantly higher accuracy than previous attacks in the same field, further highlighting website fingerprinting as a genuine threat to web privacy. We test our attack under a large open-world experimental setting, where the client can visit pages that the attacker is not aware of. We found that our new attack is much more accurate than previous attempts, especially for an attacker monitoring a set of sites with low base incidence rate. We can correctly determine which of 100 monitored web pages a client is visiting (out of a significantly larger universe) at an 85% true positive rate with a false positive rate of 0.6%, compared to the best of 83% true positive rate with a false positive rate of 6% in previous work.

To defend against such attacks, we need provably effective defenses. We show how simulatable, deterministic defenses can be provably private, and we show that bandwidth overhead optimality can be achieved for these defenses by using a supersequence over anonymity sets of packet sequences. We design a new defense by approximating this optimal strategy and demonstrate that this new defense is able to defeat any attack at a lower cost on bandwidth than the previous best.

1 Introduction

Privacy technologies are becoming more popular: Tor, a low-latency anonymity network, currently has 500,000 daily users and the number has been growing [21]. However, users of Tor are vulnerable to *website fingerprinting* attacks [4, 17, 23]. Users of other privacy technologies such as SSH tunneling, VPNs and IPsec are also vulnerable to website fingerprinting [10].

When a client browses the web, she reveals her destination and packet content to intermediate routers, which are controlled by ISPs who may be susceptible to malicious attackers, eavesdroppers, and legal pressure. To protect her web-browsing privacy, the client would need to encrypt her communication traffic and obscure her destinations with a proxy such as Tor. Website fingerprinting refers to the set of techniques that seek to re-identify these clients' destination web pages by passively observing their communication traffic. The traffic will contain packet lengths, order, and timing information that could uniquely identify the page, and website fingerprinting attacks use machine classification to extract and use this information (see Section 2).

A number of attacks have been proposed that would compromise a client's expected privacy, and defenses have been proposed to counter these attacks (see Section 3). Most previous defenses have been shown to fail against more advanced attacks [4, 6, 15]; this is because they were evaluated only against specific attacks, with no notion of provable effectiveness (against all possible attacks). In this paper, we will show an attack that further highlights the fact that clients need a provably effective defense, for which an upper bound on the accuracy of any possible attack can be given. We will then show how such a defense can be constructed. Only with a provably effective defense can we be certain that clients are protected against website fingerprinting.

The contributions of our paper are as follows:

1. We propose a significantly improved attack that achieves a higher accuracy with a training and testing time that is orders of magnitude lower than the previous best. Our attack is a k-Nearest Neighbour classifier applied on a large feature set with weight adjustment. Our attack is designed to find flaws in defenses and achieve high success rates even with those defenses, and we demonstrate that

several known defenses have almost no impact on our attack. We describe this attack in Section 4.

2. Using this attack, we tackle a large open-world problem, in which the attacker must determine which of 100 monitored pages the client is visiting, but the client can visit a large number of pages that the attacker cannot train on. We demonstrate that the attack is still truly effective in this realistic scenario in Section 5, and that it outperforms the previous best attack by Wang and Goldberg [23] (which we call OSAD) on the same data set.

3. We show that simulatable, deterministic defenses can be turned into provably private defenses in our model. In our model, we consider a defense to be successful only if it produces packet sequences that are identical (in time, order, and packet lengths) to packet sequences from different web pages. This strong notion of indistinguishability of packet sequences yields our provably private defenses. We found that the bandwidth-optimal simulatable, deterministic defense is to transmit packets using supersequences over anonymity sets. We construct a principled defense using an approximation of the smallest common supersequence problem and clustering techniques in Section 6 and evaluate it in Section 7.

We follow up with a discussion on realistic applicability and reproducibility of our results in Section 8 and conclude in Section 9.

2 Basics

2.1 Website Fingerprinting on Tor

Website fingerprinting (WF) refers to the process of attempting to identify a web-browsing client's behaviour—specifically, which web pages she is visiting—by observing her traffic traces. We assume that the client is using a proxy to hide her true destination, as well as encryption to hide her packet contents, as without these basic defenses she reveals her destination to a trivial eavesdropper. Users of Tor have these defenses.

More recent attacks can successfully perform website fingerprinting with an attacker that only has local observation capacity; i.e. the attacker merely observes the traffic traces of the client without any interference. The attacker is located on the client's network, such as the client's ISP, or he has gained control of some router near the client. Attacks requiring more capabilities have been proposed, such as attacks which leverage active traffic-shaping strategies [8], remote ping detection [9] and, sometimes, involve tampering with the client's device [12]. Our attack achieves high accuracy with only a local, passive attacker.

In general, the attacker's strategy is as follows. The attacker collects packet traces from several web pages that he is interested in monitoring. Then, the attacker observes packet traces generated by the client during her web browsing, and compares these traces with the ones he collected by performing supervised classification. We note two assumptions that all previous works on WF have made of the attacker:

1. Well-defined packet traces. It is assumed that the attacker knows where the packet trace of a single page load starts and ends. If the client takes much longer to load the next page after the current one is loaded, this assumption can be justified.

2. No other activity. We assume the client is not performing any other activity that could be confused for page-loading behaviour, such as downloading a file.

These assumptions are used by all previous works on WF as they simplify the problem, though it should be noted that these assumptions are advantageous for the attacker. We discuss how the attacker can carry out a successful attack without these assumptions in Section 8.

Website fingerprinting is harder on Tor than simple SSH or VPN tunneling [10]. This is because Tor uses cell padding, such that data is sent in fixed-size (512-byte) cells. In addition, Tor has background noise (circuit construction, SENDME packets, etc.) which interferes with website fingerprinting [23]. As Tor has a large user base and an extensive architecture upon which defenses can be applied, recent works and our work are interested in attacking and defending Tor, especially as Tor developers remain unconvinced that website fingerprinting poses a real threat [19].

2.2 Classification

Given a packet sequence, the attacker learns the client's destination web page with a classification algorithm (classifier). The attacker first gathers packet sequences of known pages that he is interested in monitoring (the training set). This is known as supervised training as the true labels of these packet sequences are known to the attacker. We can test the effectiveness of such a classifier by applying it to a data set of packet sequences that the attacker did not train on (the testing set), and measuring the accuracy of the classifier's predictions.

Central to the classifier is a notion of distance between packet sequences. A larger distance indicates that the two packet sequences are less likely to be from the same

page. Previous authors have used varying formulae for distance, ranging from comparing the occurrence counts of unique packet lengths to variations of Levenshtein distance. The distance used reflects how features are used to distinguish web pages. These features are, explicitly or implicitly, extracted from packet sequences to compare them with each other.

Our attack is based on the important observation that a class representing a web page is multi-modal. Several factors cause a web page to vary: network conditions, random advertisements and content, updating data over time, and unpredictable order of resources. Client configuration may also affect page loading.[1] An attacker can deal with multi-modal data sets by gathering enough data to have representative elements from each mode. For example, an attacker can gather two modes of a page, one for low-bandwidth connections, and another for high-bandwidth connections.[2] We use a classifier designed for multi-modal classes, for which different modes of the class do not need to have any relationship with each other.

3 Related Work

This section surveys the related work on website fingerprinting (WF). We classify attacks into those which depend on revealed resource lengths (HTTP 1.0), revealed packet lengths (HTTP 1.1, VPNs, SSH tunneling, etc.), and hidden packet lengths (Tor). We also survey the previous work on defenses in this section.

3.1 Resource length attacks

In HTTP 1.0, web page resources (images, scripts, etc.) are each requested with a separate TCP connection. This implies that an attacker who is able to distinguish between different connections can identify the total length of each resource. The earliest attacks were performed in this scenario: Cheng et al. in 1998 [5], Sun et al. in 2002 [20], and Hintz in 2003 [11]. These works showed that observing resource lengths can help identify a page. HTTP 1.1 uses persistent connections, and therefore more recent browsers and privacy technologies are not susceptible to resource length attacks.

3.2 Unique packet length attacks

Liberatore and Levine in 2006 [14] showed how unique packet lengths are a powerful WF feature with two attacks: one using the Jaccard coefficient and another using the Naïve Bayes classifier. Under the first attack, the classifier mapped each packet sequence to its set of unique packet lengths (discarding ordering and frequency). Then, it used the Jaccard coefficient as a measurement of the distance between two packet sequences. The Naïve Bayes classifier used packet lengths and their occurrence frequencies as well, but also discarded ordering and timing. The Naïve Bayes assumption is that the occurrence probabilities of different packet lengths are independent of each other. Later, Herrmann et al. [10] proposed a number of improvements to this attack by incorporating techniques from text mining.

Bissias et al. in 2006 [2] published an attack based on cross-correlation with interpacket timings, but it is less accurate than the Naïve Bayes attacks. Lu et al. in 2010 [15] published an attack that heavily focuses on capturing packet burst patterns with packet ordering, discarding packet frequencies and packet timing.

3.3 Hidden packet length attacks

Herrmann et al. were not able to successfully perform WF on Tor [17], where unique packet lengths are hidden by fixed-size Tor cells. In 2009, Panchenko et al. [17] showed an attack that succeeded against web-browsing clients that use Tor. As unique packet lengths are hidden on Tor, Panchenko et al. used other features, which are processed by a Support Vector Machine (SVM). These features attempted to capture burst patterns, main document size, ratios of incoming and outgoing packets, and total packet counts, which helped identify a page. Dyer et al. in 2012 [6] used a similar but smaller set of features for a variable n-gram classifier, but their classifier did not perform better in any of the scenarios they considered.

Cai et al. in 2011 improved the accuracy of WF on Tor. Using the edit distance to compare packet sequences, they modified the kernel of the SVM and showed an attack with significantly increased accuracy on Tor [4]. Wang and Goldberg in 2013 further improved the accuracy of Cai et al.'s scheme on Tor by modifying the edit distance algorithm [23], creating OSAD. These modifications were based on observations on how web pages are loaded. As it is the current state of the art under the same attack scenario, we will compare our attack to OSAD.

3.4 Defenses

Defenses are applied on the client's connection in order to protect her against website fingerprinting attacks. We present a new classification of defenses in this section. First, defenses can be "simulatable" or "non-simulatable". A simulatable defense can be written as a defense function D that takes in a packet sequence and

[1] On the Tor Browser changing the browser configuration is discouraged as it makes browser fingerprinting easy.

[2] Data collection on Tor will naturally result in such a situation because of random circuit selection.

outputs another packet sequence. The function does not look at the true contents of the packets, but only their length, direction and time. An advantage of simulatable defenses is the implementation cost, as non-simulatable defenses would need to be implemented on the browser and would have access to client data, which may be difficult for some clients to accept. The implementation of a simulatable defense requires no more access to information than a website fingerprinting attacker would typically have.

Secondly, defenses can be "deterministic" or "random"—for deterministic defenses the function D always returns the same packet sequence for each input packet sequence p.[3] Our goal is to design a provably private defense that has an upper bound on the accuracy of any attack. Random defenses (noise) have the disadvantage that choosing a good covering is not guaranteed. An attacker that can link together different page loads can partially remove the noise. Furthermore, implementations of random defenses must be careful so that noise cannot be easily distinguished from real packets.

Non-simulatable, random: This includes Tor's request order randomization defense. Responding to Panchenko's attack, Tor developers decided to enable pipelining on Tor and randomize pipeline size and request orders [18]. The randomization was further increased after OSAD [19]. We test our attack against the more randomized version that is built into Tor Browser Bundle 3.5.

Non-simulatable, deterministic: This includes portions of HTTPOS [16]. The HTTPOS defense is built into the client's browser, allowing the client to hide unique packet lengths by sending an HTTP range request strategically.

Simulatable, random: This includes traffic morphing [24], which allows a client to load a web page using a packet size distribution from a *different* page, and Panchenko's background noise [17], where a decoy page is loaded simultaneously with the real page to hide the real packet sequence.

Simulatable, deterministic: This includes packet padding, which is done on Tor, and BuFLO, presented and analyzed by Dyer et al. [6]. BuFLO sends data at a constant rate in both directions until data transfer ends. In this work, we will show that defenses in this category can be made to be provably private,[4] and we will show such a defense with a much lower overhead than BuFLO.

[3] Using a random procedure to learn D does not make D itself random.

[4] BuFLO is not provably private on its own.

4 Attack

In this section, we describe our new attack, which is designed to break website fingerprinting defenses. Our attack is based on the well-known k-Nearest Neighbours (k-NN) classifier, which we briefly overview in Section 4.1. The attack finds flaws in defenses by relying on a large feature set, which we describe in Section 4.2. We then train the attack to focus on features which the defense fails to cover and which therefore remain useful for classification. We describe the weight adjustment process in Section 4.3.

4.1 k-NN classifier

k-NN is a simple supervised machine learning algorithm. Suppose the training set is S_{train} and the testing set is S_{test}. The classifier is given a set of training points (packet sequences) $S_{train} = \{P_1, P_2, \ldots\}$. The training points are labeled with classes (the page the packet sequence was loaded from); let the class of P_i be denoted $C(P_i)$. Given a testing point $P_{test} \in S_{test}$, the classifier guesses $C(P_{test})$ by computing the distance $D(P_{test}, P_{train})$ for each $P_{train} \in S_{train}$. The algorithm then classifies P_{test} based on the classes of the k closest training points.

Despite its simplicity, the k-NN classifier has a number of advantages over other classifiers. Training involves learning a distance between pairs of points; the classifier could use a known (e.g. Euclidean) distance, though selecting the distance function carefully can greatly improve the classification accuracy. Testing time is very short, with a single distance computation to each training point. Multi-modal sets can be classified accurately, as the classifier would only need to refer to a single mode of each training set.

The k-NN classifier needs a distance function d for pairs of packet sequences. The distance is non-trivial for packet sequences. We want the distance to be accurate on simple encrypted data without extra padding, but also accurate when defenses are applied that remove features from our available feature set. We therefore start with a large feature set $F = \{f_1, f_2, \ldots\}$. Each feature is a function f which takes in a packet sequence P and computes $f(P)$, a non-negative number. Conceptually, each feature is designed such that members of the same class are more likely to have similar features than members of different classes. We give our feature set in Section 4.2. The distance between P and P' is computed as:

$$d(P,P') = \sum_{1 \le i \le |F|} w_i |f_i(P) - f_i(P')|$$

The weights $W = \{w_1, w_2, \ldots, w_{|F|}\}$ are learned as in Section 4.3, where we describe how the weights for uninformative features (such as one that a defense success-

fully covers) are reduced. As weight learning proceeds, the k-NN distance comes to focus on features that are useful for classification.

We tried a number of other distances, including the edit distance used by Cai et al. [4] and OSAD, which they used to compute the kernel of SVMs. The results for using their distances on the k-NN classifier are similar to those using the SVM. As we shall see in Section 5, using our proposed distance allows a significant improvement in accuracy over these distances, with or without extra defenses.

4.2 Feature set

Our feature set is intended to be diverse. The construction of the feature set is based on prior knowledge of how website fingerprinting attacks work and how defenses fail.

Our feature set includes the following:

- General features. This includes total transmission size, total transmission time, and numbers of incoming and outgoing packets.

- Unique packet lengths. For each packet length between 1 and 1500, and each direction, we include a feature which is defined as 1 if it occurs in the data set and 0 if it does not. This is similar to the algorithms used by Liberatore and Levine [14] and Herrmann et al. [10], where the presence of unique packet lengths is an important feature. These features are not useful when packet padding is applied, as on Tor.

- Packet ordering. For each outgoing packet, we add, in order, a feature that indicates the total number of packets before it in the sequence. We also add a feature that indicates the total number of incoming packets between this outgoing packet and the previous one. This captures the burst patterns that helped Cai et al. achieve their high accuracy rates.

- Concentration of outgoing packets. We count the number of outgoing packets in non-overlapping spans of 30 packets, and add that as a feature. This indicates where the outgoing packets are concentrated without the fineness (and volatility) of the packet ordering features above.

- Bursts. We define a burst of outgoing packets as a sequence of outgoing packets, in which there are no two adjacent incoming packets. We find the maximum and mean burst length, as well as the number of bursts, and add them as features.

- Initial packets. We also add the lengths of the first 20 packets (with direction) in the sequence as features.

Some feature sets, such as packet ordering, have variable numbers of features. We define a maximum number of features for the set, and if the packet sequence does not have this many features, we pad with a special character (X) until it reaches the maximum number. Recall that our distance is the weighted sum of absolute differences between features; let us denote the difference as $d_{f_i}(P,P')$. For each feature f_i, if at least one of the two values is X, then we define $d_{f_i}(P,P')$ to be 0, such that the difference is ignored and does not contribute to the total distance. Otherwise, we compute the difference as usual.

We treat all features equally. However, we note that as the general features are amongst the strongest indicators of whether or not two packet sequences belong to the same mode of a page, we could use them with a search algorithm to significantly reduce training and testing time (i.e. reject pages with significantly different values in the general feature without computing the whole distance).

The total number of features is close to 4,000 (3,000 of which are just for the unique packet lengths). If a defense covers some features and leaves others open (e.g. traffic morphing retains total transmission size and burst features), our algorithm should be successful in adjusting weights to focus on useful features.

We design our attack by drawing from previous successful attacks, while allowing automatic defense-breaking. In particular, we note that there exists a choice of weights for which our attack uses a similar distance metric as the attacks proposed by Cai et al. [4] and Wang and Goldberg [23], as well as the Jaccard coefficient by Liberatore and Levine [14]. However, we will find better choices of weights in the next subsection. We drew the inspiration for some features from the work by Panchenko et al. [17], in particular, those concerning the start of the page (which may indicate the size of the HTML document). We note that unlike Panchenko et al. [17], we do not add the entire packet sequence as features.

4.3 Weight initialization and adjustment

In this subsection, we describe how we learn $w_1, w_2, \ldots, w_{|F|}$, the weights that determine our distance computation. The values of these weights determine the quality of our classifier. We learn the weights using an iterative, local weight-learning process as follows. The weight-learning process is carried out for R rounds (we will see how the choice of R affects the accuracy later). For each round, we focus on a point $P_{train} \in S_{train}$ (in or-

der), performing two steps: the weight recommendation step and the weight adjustment step.

Weight recommendation. The objective of the weight recommendation step is to find the weights that we want to reduce. During the weight recommendation step, the distances between P_{train} and all other $P' \in S_{train}$ are computed. We then take the closest k_{reco} points (for a parameter k_{reco}) within the same class $S_{good} = \{P_1, P_2, \ldots\}$ and the closest k_{reco} points within all other classes $S_{bad} = \{P'_1, P'_2, \ldots\}$; we will focus only on those points.

We denote $d(P,S)$, where S is a set of packet sequences, as the sum of the distances between P and each sequence in S.

Let us denote

$$d_{maxgood_i} = \max(\{d_{f_i}(P_{train}, P) | P \in S_{good}\})$$

For each feature, we compute the number of relevant bad distances, n_{bad_i}, as

$$n_{bad_i} = |\{P' \in S_{bad} | d_{f_i}(P_{train}, P') \leq d_{maxgood_i}\}|$$

This indicates how bad feature f_i is in helping to distinguish S_{bad} from S_{good}. A large value of n_{bad_i} means that feature f_i is not useful at distinguishing members of P_{train}'s class from members of other classes, and so the weight of f_i should be decreased; for example, features perfectly covered by a defence (such as unique packet lengths in Tor) will always have $n_{bad_i} = k_{reco}$, its maximum possible value. Conversely, small values of n_{bad_i} indicate helpful features whose weights should be increased.

Weight adjustment. We adjust the weights to keep $d(P_{train}, S_{bad})$ the same while reducing $d(P_{train}, S_{good})$. Then, for each i such that $n_{bad_i} \neq \min(\{n_{bad_1}, n_{bad_2}, \ldots, n_{bad_{|F|}}\})$, we reduce the weight by $\Delta w_i = w_i \cdot 0.01$. We then increase all weights w_i with $n_{bad_i} = \min(\{n_{bad_1}, n_{bad_2}, \ldots, n_{bad_{|F|}}\})$ equally such that $d(P_{train}, S_{bad})$ remains the same.

We achieved our best results with two more changes to the way weights are reduced, as follows:

- We further multiply $\Delta w_i = w_i \cdot 0.01$ by n_{bad_i}/k_{reco}. Therefore, a weight with greater n_{bad_i} (a less informative weight) will be reduced more.

- We also decrease Δw_i if P_{train} is already well classified. N_{bad} is defined as:

  $$N_{bad} = |\{P' \in S_{bad} | d(P_{train}, P') \leq d_{maxgood}\}|$$

  Specifically, we multiply Δw_i by $0.2 + N_{bad}/k_{reco}$. N_{bad} can be considered an overall measure of how poorly the current point is classified, such that points which are already well-classified in each iteration have less of an impact on the weights. The addition of 0.2 indicates that even perfectly classified points still have some small impact on the weights (so that the weight adjustment will not nullify their perfect classification).

Both of these above changes improved our classification accuracy. We achieved our best results with $k_{reco} = 5$.

We initialized the weight vector W randomly by choosing a random value for each w_i uniformly between 0.5 and 1.5. Adding randomness gave us a chance of finding better solutions than a deterministic algorithm as we could avoid local maxima that bind our classifier away from the global maximum.

Note that we are not claiming these particular choices of parameters and constants yield an optimal attack, and further work may yet uncover improved attacks against defenses without provable privacy guarantees.

5 Attack evaluation

Our attack is specifically designed to find gaps in defenses, and in this section we will demonstrate its efficacy with experimentation on real web traffic. We will first begin by showing the effectiveness of our scheme against Tor with its default packet padding and order randomization defense in Section 5.1. This setting is a good standard basis of comparison as WF is a threat to the privacy guarantees provided by Tor, and several of the latest state-of-the-art attacks are designed for and evaluated on Tor. We will see that our attack performs better than the best known attacks. The parameters of our attack can be modified to decrease the false positive rate at the cost of decreasing the true positive rate, and we examine the tradeoff in Section 5.2. Then, we show that our attack is also more powerful than known attacks on various known and published defenses in Section 5.3, with a number of defenses shown to be nearly completely ineffective against our scheme.

5.1 Attack on Tor

We validate our attack in two experimental settings to demonstrate the effectiveness of our attack on Tor.

First, we perform experiments in an open-world experimental setting. Even though the number of pages in the world wide web is far too large for us to train on, we can achieve realistic results by limiting the objective of the attacker. Here, the attacker wants to decide whether or not a packet sequence comes from a monitored page; additionally, for monitored pages, the attacker aims to identify the page. We denote the non-monitored page set

that the attacker uses for training as C_0, and the effects of varying its size will be tested in evaluation. This open-world experimental setting gives us realistic results for plausible attackers.

We use a list of 90 instances each of 100 sensitive pages as well as 1 instance each of 5,000 non-monitored pages. We note that this problem is more difficult for the attacker than any that has been evaluated in the field, as other authors have evaluated their schemes on either strictly closed-world settings or very small open-world problems (a few monitored pages). It is a realistic goal for the attacker to monitor a large set of pages in the open-world setting.

Our list of 100 monitored pages was compiled from a list of blocked web pages from China, the UK, and Saudi Arabia. These include pages ranging from adult content, torrent trackers, and social media to sensitive religious and political topics. We selected our list of 5,000 non-monitored pages from Alexa's top 10,000 [1], in order, excluding pages that are in the list of monitored pages by domain name. The inherent differences between the above data sets used for training and testing assist classification, just as they would for a realistic attacker. Page loading was done with regular circuit resetting, no caches and time gaps between multiple loads of the same page (as suggested by Wang and Goldberg [23]), such that the attacker will not use the same circuits as the target client, or collect its data at the same time. We used iMacros 8.6.0 on Tor Browser 3.5.1 to collect our data.

Training the k-Nearest Neighbour classifier is required to learn the correct weights. We learn the weights by splitting part of the training set for weight adjustment and evaluation as above. We perform weight adjustment for $R = 6000$ rounds on 100 pages and 60 instances each, which means that every instance is cycled over once. Then, accuracy is computed over the remaining 30 instances each, on which we perform all-but-one cross validation. The use of cross validation implies that the attacker will never train on the same non-monitored pages that the client visits.

For our attack, we decided that a point should be classified as a monitored page only if all k neighbours agree on which page it is, and otherwise it will be classified as a non-monitored page. This helped reduce false positives at a relatively small cost to the true positives. We vary the number of neighbours k from 1 to 15 as well as the number of non-monitored training pages $|C_0|$ used from 10 to 5000,[5] and we show our results in Figure 1. We measure the True Positive Rate (TPR), which is the probability that a monitored page is correctly classified as that particular monitored page, and the False Positive Rate (FPR), which is the probability that a non-monitored page is incorrectly identified as being monitored.[6] We can achieve TPR 0.85 ± 0.04 for FPR 0.006 ± 0.004, or respectively TPR 0.76 ± 0.06 for FPR 0.001 ± 0.001.

[5] We note that the choice of $|C_0|$ does *not* represent a world with fewer pages available to the client—it is the attacker's decision on how much he wishes the bias towards non-monitored sites to be. The visited sites are always drawn from Alexa's top 10,000.

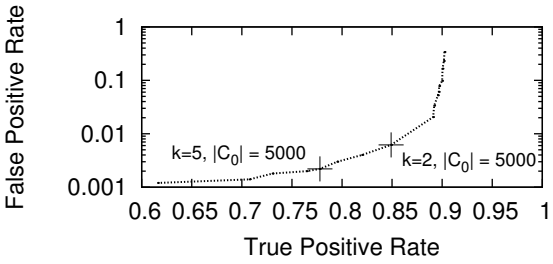


Figure 1: Performance of our attack while varying the attack parameters k and $|C_0|$. Only the y-axis is logarithmically scaled.

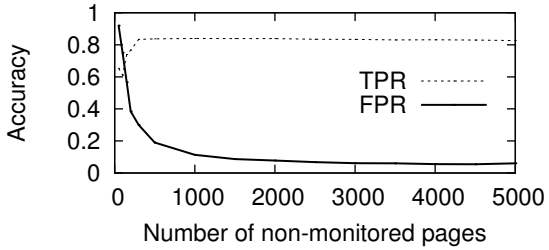


Figure 2: True Positive Rate and False Positive Rate changes in OSAD [23] as the open set size increases. There is almost no change in either value after $|C_0| > 2500$.

We compare these values to OSAD, which we apply to our data set as well, and show the results in Figure 2. Increasing the number of non-monitored pages $|C_0|$ increases TPR and reduces FPR. After $|C_0| > 2500$, we could not see a significant benefit in adding more elements. At $|C_0| = 5000$, the classifier achieves a TPR of 0.83 ± 0.03 and a FPR of 0.06 ± 0.02.

We see that OSAD cannot achieve FPR values nearly as low as ours, and it may be considered impractical for the attacker to monitor large sets in the open-world setting with the old classifier, especially if the base incidence rate is low. For example, if the base incidence rate of the whole sensitive set is 0.01 (99% of the time the client is visiting none of these pages), and our new classifier claims to have found a sensitive site, the decision is correct at least 80% of the time, the rest being false positives. For Wang and Goldberg's classifier, the same value would be about 12%. The difference is further exacerbated with a lower base incidence rate, which may be realistic for particularly sensitive web sites.

[6] If a monitored page is incorrectly classified as a *different* monitored page or as a non-monitored page, it is a false negative.

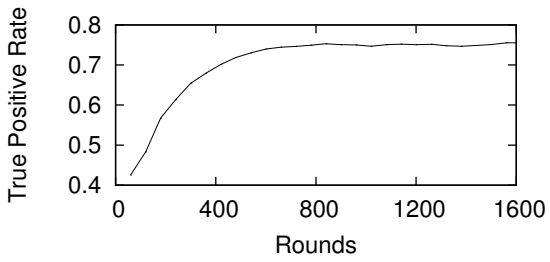


Figure 3: TPR when varying the number of rounds used for training our attack, with $k = 5$ and $|C_0| = 500$. FPR is not shown because there is very little change over time.

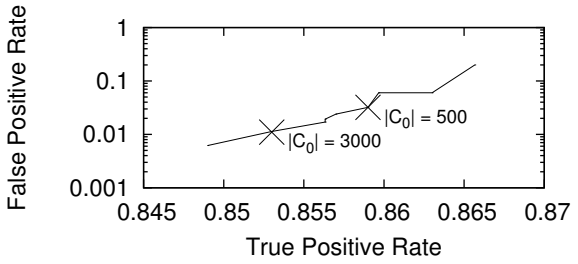


Figure 4: Results for FPR vs. TPR while varying bias towards non-monitored data set C_0. $k = 2$.

The training and testing time for our classifier (weight adjustment) is very small compared to previous state-of-the-art classifiers. The number of rounds, R, determines the quality of our weights. We show in Figure 3 how the true positive rate changes with R on $|C_0| = 500$ non-monitored sites and $k = 5$ neighbours. We see that the accuracy levels off at around 800 rounds, and did not drop up to 30,000 rounds.

The weight training time scales linearly with R and also scales linearly with the number of instances used for weight training. The training time is around $8 \cdot 10^{-6} \cdot |S_{train}| \cdot R$ CPU seconds, measured using a computing cluster with AMD Opteron 2.2 GHz cores. This amounts to around 120 CPU seconds for 1000 rounds in our set with $|C_0| = 5000$. This can be compared to around 1600 CPU hours on the same data set using OSAD and 500 CPU hours using that of Cai et al. Training time also scales quadratically with the number of training instances with these previous classifiers.

The testing time amounts to around 0.1 CPU seconds to classify one instance for our classifier and around 800 CPU seconds for OSAD, and 450 CPU seconds for Cai et al. The testing time per instance scales linearly with the number of training elements for all three classifiers. We can reduce the training and testing time for our classifier further by around 4 times if we remove the unique packet length features, which are useless for Tor cells.

We also perform experiments on the closed-world experimental setting. Under the closed-world experimental setting, the client does not visit non-monitored pages. We use the same data set of sensitive pages as above

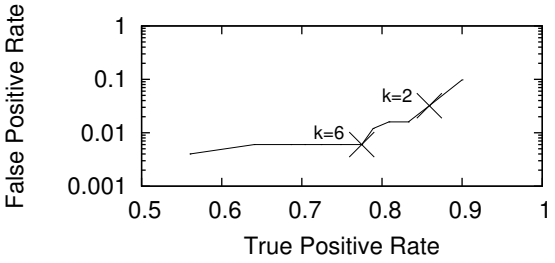


Figure 5: Best results for FPR vs. TPR while varying number of neighbours k. $|C_0| = 500$.

for these experiments. Although the closed-world setting does not carry the same realistic implications as the open-world setting, it focuses attention on the ability of the classifier to distinguish between pages and it has been a useful basis of comparison in the field. We also tested our classifier on the data set used by Wang and Goldberg to facilitate a more direct comparison, and the accuracy was 0.95 ± 0.02 compared to 0.91 ± 0.06 for OSAD and 0.88 ± 0.03 for Cai et al. We also compared them on our new data set, and the accuracy was 0.91 ± 0.03 for ours and 0.90 ± 0.02 for OSAD. There appears to be no significant difference in the closed-world scenario, although the superior accuracy of our classifier under the realistic open-world scenario is clear.

5.2 Training confidence

The numbers for true and false positive rates as shown above may not be desirable for some cases. The optimal numbers depend on the expected base rate of the monitored activity as well as the application intended by the attacker. Parameters of our attack can be adjusted to increase true positive rate at the cost of increasing false positive rate, or vice versa.

We can vary the size of the non-monitored training page set to affect accuracy as our implementation of the k-Nearest Neighbour classifier is susceptible to bias towards larger classes. We fix the number of neighbours at $k = 2$, vary the number of non-monitored training pages $|C_0|$ from 10 to 5000 and show the results in Figure 4.

We can also vary k, the number of neighbours. We fix the number of non-monitored pages, $|C_0|$, at 500, and vary k from 1 to 15, showing the results in Figure 5. Decreasing $|C_0|$ and decreasing k each increases both true positives and false positives.

We can see that varying the number of neighbours used is much more important for determining TPR than varying the size of C_0, the set of non-monitored pages. In fact, almost all of the graph in Figure 1 can be drawn only by varying k with $|C_0| = 5000$, suggesting that it is advantageous for the attacker to have a large number of non-monitored training pages.

Table 1: Accuracy of our attack on various defenses. Closed-world simulation is used to enable comparison with previous known results.

Defense	Accuracy	Bandwidth overhead
Traffic morphing [24]	0.82 ± 0.06	$50\% \pm 10\%$
HTTPOS split [16]	0.86 ± 0.03	$5.0\% \pm 0.6\%$
Decoy pages [17]	0.30 ± 0.06	$130\% \pm 20\%$
BuFLO [6]	0.10 ± 0.03	$190\% \pm 20\%$

5.3 Attack on Other Defenses

Our attack is specifically designed to break WF defenses that leave features open for classification. The analysis in previous sections was performed on Tor packets, which already uses padding, pipelining and order randomization. We add further defenses on top of Tor's defenses. The list of defenses we evaluate in this section are as follows:

- Traffic morphing [24]. Traffic morphing maps packet sizes from one site to a packet distribution drawn from another site, in an attempt to mimic the destination site. In our implementation, each site attempted to mimic `google.com` as it is reasonable to assume that the client wishes to mimic the most popular page.
- HTTPOS split [16]. Although HTTPOS has a large number of features, one of its core features is a random split on unique packet lengths by cleverly utilizing HTTP range requests. We analyze HTTPOS by splitting incoming packets and also padding outgoing packets.[7]
- Panchenko's decoy pages [17]. As a defense against their own attack, Panchenko et al. suggested that each real page should be loaded with a decoy page. We chose non-monitored pages randomly as decoy pages.
- BuFLO [6]. Maximum size packets are sent in both directions at equal, constant rates until the data has been sent, or until 10 seconds have passed, whichever is longer.

We implement these defenses as simulations. For Panchenko's noise and BuFLO we implement them using Tor cells as a basic unit in order to reduce unnecessary overhead from these defenses when applied on Tor. We assume that the attacker is aware of these defenses and collects training instances on which the defense is applied; this is realistic as the above defenses are all distinctive and identifiable.

We apply our attack, and show the results in Table 1. This can be compared to a minimum accuracy of 0.01 for random guessing. We see that even with large overhead, the defenses often fail to cover the page, and our attack always performs significantly better than random guessing. For BuFLO, our particular data set gave a larger overhead than previous work [22] because most packet sequences could be loaded within 10 seconds and therefore required end-of-sequence padding to 10 seconds. In particular, traffic morphing and HTTPOS split have almost no effect on the accuracy of our attack.

6 Defense

In this section, we design a provably private defense—a defense for which there exists an upper bound on the accuracy of any attack (given the data set). As Tor is bandwidth-starved [21], we attempt to give such a defense with the minimum bandwidth cost. This is an extension of the idea proposed by Wang and Goldberg [22] for their defense, Tamaraw.

In Section 6.1, we first show how such an upper bound can be given for *simulatable, deterministic defenses*—that is, this class of defenses can be made to be provably private. We then show in Section 6.2 that the optimal defense strategy (lowest bandwidth cost) in such a class is to compute supersequences over sets of packet sequences (anonymity sets). We try to approximate the optimal defense strategy, by describing how these sets can be chosen in Section 6.3, and how the supersequence can be estimated in Section 6.4.

6.1 Attacker's upper bound

We describe how we can obtain an upper bound on the accuracy of any attack given a defended data set. The attacker, given an observation (packet sequence) p, wishes to find the class it belonged to, $C(p)$.

To calculate the maximum success probability given the testing set, we assume the greatest possible advantage for the attacker. This is where the attacker is allowed to train on the testing set.[8] In this case the attacker's optimal classification strategy is to record the true class of each observation, $(p, C(p))$. The attacker will only ever make an error if the same observation is mapped to several different classes, which are indistinguishable for the observation. We denote the possibility set of p as the multiset of classes with the same observation p, $Q(p) =$

[7] HTTPOS has been significantly modified by its authors since their original publication, in part due to the fact that Cai et al. were able to break it easily [4].

[8] Our testing set is in fact a multiset as repeated observation-class pairs are possible.

$\{C_1, C_2, \ldots\}$ ($C(p) \in Q(p)$), where the occurrence count of a class is the same as in the testing set with observation p.

The attacker's optimal strategy is to find the class C_{max} that occurs the most frequently for the same observation p, and during classification the attacker will return C_{max} for the observation p. This will induce an accuracy value upon p:

$$Acc(p) = \frac{|\{C \in Q(p) | C = C_{max}\}|}{|Q(p)|}$$

This method returns the best possible accuracy for a given testing set as it makes the lowest possible error for observations mapping to multiple classes.

Cai et al. [3] have proposed two different ways to denote the overall accuracy of a set of packet sequences:

- Non-uniform accuracy. This is the mean of accuracies $Acc(p)$ for $p \in S_{test}$.
- Uniform accuracy. This is the maximum accuracy $Acc(p)$ for $p \in S_{test}$

Tamaraw can only achieve non-uniform accuracy. In this work, we design a defense for uniform accuracy, but the defense can be extended to other notions as well. While we will use different sets to train our defense and test it on client behaviour, we will say that the defense has a maximum uniform accuracy as long as it does so on the training set (as it is always possible to construct a testing set on simulatable, deterministic defenses on which at least one page has an accuracy of 1). A defense that achieves a maximum uniform accuracy of A_u automatically does so for non-uniform accuracy, but not vice-versa. In the following we work with a uniform prior on a fixed-size testing set to facilitate comparison with previous work.

6.2 Optimal defense

In this section, we show the bandwidth-optimal simulatable, deterministic defense. As we work with Tor cells, in the following a packet sequence can be considered a sequence of -1's and 1's (downstream and upstream packets respectively), which is useful for hiding unique packet lengths [22]. We say that sequence q is a subsequence of sequence p (or that p is a supersequence of q) if there exists a set of deletions of -1 and 1 in p to make them equal (maintaining order). With abuse of notation, we say that if S is the input packet sequence multiset, then $D(S) = \{D(p) | p \in S\}$ denotes the output packet sequence multiset after application of the defense. The cost (bandwidth overhead) of $D(p)$ is $B(D(p)) = \frac{|D(p)| - |p|}{|p|}$, and similarly for a set of packet sequences the overhead is $B(D(S)) = \frac{\sum_{p \in S} |D(p)| - \sum_{p \in S} |p|}{\sum_{p \in S} |p|}$. Given S, we want to identify D such that $B(D(S))$ is minimal.

For each packet sequence p_1, let us consider the set of packet sequences that map to the same observation after the defense is applied, which we call the *anonymity set* of p_1. We write the set as $A(p_1) = \{p_1, p_2, \ldots, p_E\}$; i.e. $D(p_1) = D(p_i)$ for each i. The shortest $D(p_1)$ that satisfies the above condition is in fact the shortest common supersequence, written as $f_{scs}(A(p_1)) = D(p_i)$ for each $1 \leq i \leq E$.

In other words, the optimal solution is to apply the shortest common supersequence function to anonymity sets of input sequences. This defense can be applied with the cooperation of a proxy on the other side of the adversary; on Tor, for example, this could be the exit node of the circuit. However, finding such an optimal solution requires solving two hard problems.

Anonymity set selection. First, given the set of all possible packet sequences, we want to group them into anonymity sets such that, for a given bound on attacker accuracy, the overhead will be minimized.

The shortest common supersequence (SCS) problem. Then, we must determine the SCS of all the packet sequences in the anonymity set. This is in general NP-hard. [13]

In the next two sections we describe our solutions to the above problems.

6.3 Anonymity set selection

We note that the client is not always able to choose anonymity sets freely. For example, the client cannot easily know which anonymity set a page load should belong to before seeing the packet sequence. While the client can gain information that assists in making this decision (the URL, previous page load data, training data, information about the client network, the first few packets of the sequence, etc.), the mere storage and usage of this information carries additional privacy risks. In particular, the Tor Browser keeps no disk storage (including no cache except from memory), so that storing extraneous information puts the client at additional risk. In this section, we describe how realistic conditions impose restrictions on the power of the client to choose anonymity sets.

We formalize this observation by imposing additional limits on anonymity set selection in the defense D trained on testing set S_{test}. We define four levels of information for a client applying a simulatable, deterministic website fingerprinting defense:

Table 2: Relationship between different levels of information and how we train and test our supersequences. Under "Supersequence", we describe what supersequences we would use at this level of information. Clustering is done if we want multiple supersequences.

Information	Supersequence	Training and Testing
No information	One supersequence	Different sites, instances
Sequence end information	One supersequence, stopping points	Different sites, instances
Class information	Multiple supersequences, stopping points	Same sites, different instances
Full information	Multiple supersequences	Same sites, instances

1. No information. The client has no information at all about the packet sequence to be loaded. This means $A(p) = S_{test}$, that is to say all sequences map to a single anonymity set.

2. Sequence end information. The client knows when the sequence has ended, but this is the only information the client gets about the packet sequence. This means that D can only vary in length; for any p, q, such that $|D(p)| \geq |D(q)|$, then the first $|D(q)|$ packets of $D(p)$ are exactly $D(q)$, that is, we say that $D(q)$ is a prefix of $D(p)$.

3. Class-specific information. Only the identity of the page is known to the client, and the client has loaded the page before with some information about the page, possibly with realistic offline training. The client cannot distinguish between different packet sequences of the same page (even though the page may be multi-modal). This is the same as the above restriction but only applied if p and q are packet sequences from the same web page.

4. Full information. No restrictions are added to D. The client has prescient information of the full packet sequence. Beyond class-specific information, the client can gain further information by looking into the future at the contents of the packet sequence, learning about her network, and possibly using other types of extraneous information. This level is not generally of practical interest except for serving as a bound for any realistic defense.

We use clustering, an unsupervised machine learning technique, to find our anonymity sets. We show how the above levels of information affect how supersequences will be computed and how testing needs to be performed in Table 2.

Optimality under the above levels of information requires the computation of supersequences over anonymity sets. If we have only sequence end information, there is only one supersequence, and we do not need to perform clustering. Instead, possible outputs of the defense simply correspond to a prefix of the one supersequence, terminating at one of a specified set of stopping points. We find the stopping points by selecting the earliest points where our maximum uniform accuracy would be satisfied. All packet sequences sent under this defense will be padded to the next stopping point. This is similar to a strategy suggested by Cai et al. [3]

If we have class-level information, we need to perform two levels of anonymity set selection. On the first level, we cluster the packet sequences within each class to decide which supersequence the client should use. For this level of clustering, we first decide on the number of supersequences in the set. Then, we randomly choose a number of "roots" equal to this number of supersequences. We cycle over every root, assigning the closest packet sequence that has not yet been classified. For this we need to define a distance between each pair of packet sequences p and q. Suppose p' and q' are the first $min(|p|,|q|)$ packets of p and q respectively. The distance between p and q is given as $2|f_{scs}(p',q')| - |p'| - |q'|$. We use this distance to measure how different two packet sequences are, without considering their respective lengths, which would be addressed by the second level. On the second level, we find stopping points, with the same strategy as that used under sequence end information. The use of an additional first level of clustering reduces the number of stopping points available for use, given a fixed number of clusters, so that using too many clusters may in fact have a higher bandwidth overhead (see Section 7).

For full information, we perform clustering with the distance between two packet sequences p and q as $2|f_{scs}(p,q)| - |p| - |q|$. Here we select roots with evenly spread out lengths.

6.4 SCS approximation

For the SCS of two packet sequences there is an exact solution that can be found using dynamic programming; however, the SCS of multiple sequences is in general NP-hard [13].

We present a simple algorithm that approximates a solution to the shortest common supersequence problem. To approximate $f_{scs}(\{p_1, p_2, \ldots, p_n\})$, we define a counter for each packet sequence $c_1, c_2, \ldots, c_n$, which starts at 1. We count the number of sequences for which

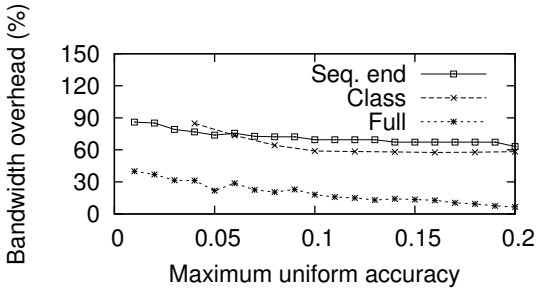


Figure 6: Bandwidth overhead for three levels of information: sequence end information (Seq. end), class-specific information (Class), and full information (Full). Using no information results in a bandwidth overhead that is much higher than that shown in the graph.

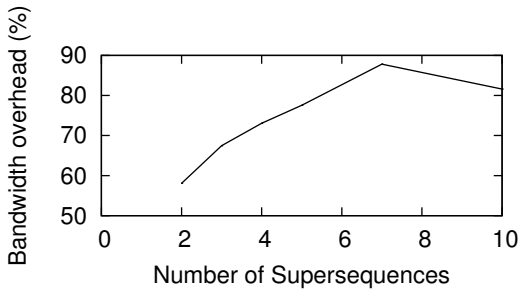


Figure 7: Bandwidth overhead for class-specific information if more than two supersequences are used, at 20 clusters. The number of stopping points available decreases, but the total number of supersequences times the number of stopping points is always at least 20.

the c_i-th element of p_i is an outgoing packet. If the number exceeds $n/4$, we append an outgoing packet to the common supersequence, and increment all c_i for which the c_i-th element of p_i is an outgoing packet by 1. Else, we append an incoming packet, and increase the corresponding counts by 1. We do this iteratively until each counter c_i becomes $|p_i|+1$, and the algorithm terminates. The choice of $n/4$ is because for web page loading, there are fewer outgoing packets than incoming packets, and this choice reduces our overhead significantly.

We note that it is easy to construct cases where the above algorithm performs very poorly. In fact, it is known that any polynomial-time approximation algorithm of shortest common supersequences cannot have bounded error [13].

7 Defense evaluation

In this section we evaluate our defense for bandwidth overhead, as well as its effectiveness in stopping our new attack.

We implemented our defenses with different levels of information as seen above. We used the same data set used to test our attacks—100 sites, 30 instances each—and attempted to protect them. The defender attempts to achieve a given maximum uniform accuracy (by determining the number of clusters or stopping points). We show the results in Figure 6. For class-level information, we used two supersequences and $N/2$ stopping points in each supersequence. We can see the full information setting has a much lower bandwidth overhead than sequence end information or class-level information. With our clustering strategy, using two supersequences under class-level information is only sometimes beneficial for the overhead. It is possible that a clever clustering strategy for class-level information could achieve lower bandwidth overheads.

For class-level information, we used two supersequences as above. It is interesting to know if increasing the number of supersequences (and correspondingly lowering the number of stopping points) will give better bandwidth overhead. In other words, we want to know if it is worth suffering greater overhead for padding to stopping points to have more finely tuned supersequences. We fix the target maximum uniform accuracy to 20%. The results are shown in Figure 7. We can see that using more than two supersequences only increases the bandwidth overhead. It is possible that if the defender can tolerate a higher maximum uniform accuracy, then it would be optimal to use more than two supersequences.

Finally, we apply our new attack to a class-level defense with a maximum uniform accuracy of 0.1, where the overhead is approximately $59\% \pm 3\%$. We achieved an accuracy of 0.068 ± 0.007. This can be compared to Table 1, where we can see that the attack achieved an accuracy of 0.30 ± 0.06 for Panchenko's decoy pages with an overhead of $130\% \pm 20\%$ and an accuracy of 0.10 ± 0.03 for BuFLO with an overhead of $190\% \pm 20\%$. Furthermore, we do not know if there exist better attacks for these defenses, but we know that no attack can achieve a better accuracy than 0.1 on our defense (using the same data set). We also compared our work with Tamaraw, which had a $96\% \pm 9\%$ overhead on the same data set for non-uniform accuracy. Our attack achieved an accuracy of 0.09 ± 0.02, although highly non-uniformly. Indeed, on 16 sites out of 100, the accuracy of the attacker was more than 0.2, and the most accurately classified site had accuracy 0.6.

8 Discussion

8.1 Realistically applying an attack

Like other website fingerprinting works in the field, we make the assumption that the attacker has an oracle that

can answer whether or not a particular sequence is generated from a single page load, and that the user does not prematurely halt the page load or perform other types of web activity. Here we discuss a few strategies to deal with possible sources of noise when applying website fingerprinting to the real world.

The attacker can use a number of signals to identify the start of a packet sequence. We found that the start of a packet sequence generally contains around three times more outgoing packets than the rest of the sequence. If the user is accessing a page for which she does not have a current connection (i.e. most likely the user is visiting a page from another domain), then the user will always send one or two outgoing connections (depending on the browser setting) to the server, followed by acceptance from the server, followed by a GET request from the main page, and then by data from the server. This particular sequence could be identifiable.

Unfortunately for Tor users, website fingerprinting is made easier due to a number of design decisions. On Tor, users are discouraged from loading videos, using torrents, and downloading large files over Tor, which are types of noise that would interfere with website fingerprinting. It is hard to change user settings on the Tor Browser; the configuration file is reset every time the Tor Browser is restarted, which implies that different Tor users have similar browser settings. As there is no disk caching, Tor users have to log in every time the Tor Browser is restarted before seeing personalized pages. For example, Facebook users on Tor must go through the front page, which has no variation and is easily identifiable. This is meant to preserve privacy from server-side browser fingerprinting attacks, but they also make website fingerprinting easier.

8.2 Realistic consequences of an attack

Here we discuss how our attack can be used realistically to break the privacy of web users. Our attack is not all-powerful; it is not likely to find a single sensitive page access among millions without error. The quality of the results depends on the base incidence rate of the client's access. With our classifier, if an attacker wishes to identify exactly which of a set of 100 pages a client is visiting, and she almost never visits those pages (less than 0.1% of page visits), then false alarms will overwhelm the number of true positives. We note that many sensitive pages have high rates of incidence as they are within Alexa's top 100 (torrent sites, adult sites, social media), especially if the client feels it necessary to use Tor.

We envision our attack as a strong source of information that becomes more powerful with the use of other orthogonal sources of information. For instance, a government agency observes that a whistleblower has released information on a web page, or that she has just posted a sensitive or incendiary article on a blog, and it is known that this whistleblower is likely to use Tor. The agency will only need to search amongst Tor streams in the last few minutes within the nation (or a smaller local area). As Tor streams are easily identifiable [7], the number of Tor users at any given moment is small enough for our accurate attack to lead to the capture of a Tor-using dissident. This strongly suggests that some sort of defense is necessary to protect the privacy of web clients.

8.3 Reproducibility of our results

To ensure reproducibility and scientific correctness, we publish the following: [9]

- The code for our new attack. This includes our feature set, parameters used for our weight learning process, and a number of weight vectors we learned which succeeded at classification against specific defenses, including the Tor data set.
- The code for our new defense. This includes the clustering strategy and the computation for stop points, as well as the supersequences we eventually used to achieve the results in this paper.
- Our implementations of known attacks and defenses, which we compared and evaluated against ours.
- The data sets we used for evaluation. This includes the list of monitored and non-monitored sites we visited over Tor, and the TCP packets we collected while visiting those sites and which we processed into Tor cells. We also include the feature vectors we computed over this data set.

9 Conclusion

In this work, we have shown that using an attack which exploits the multi-modal property of web pages with the k-Nearest Neighbour classifier gives us a much higher accuracy than previous work. We use a large feature set and learn feature weights by adjusting them based on shortening the distance towards points in the same class, and we show that our procedure is robust. The k-NN costs only seconds to train on a large database, compared to hundreds of hours for previous state-of-the-art attacks. The attack further performs well in the open-world experiments if the attacker chooses k and the bias towards non-monitored pages properly. Furthermore, as

[9] They can be found at `https://crysp.uwaterloo.ca/software/webfingerprint/`

the attack is designed to automatically converge on unprotected features, we have shown that our attack is powerful against all known defenses.

This indicates that we need a strong, provable defense to protect ourselves against ever-improving attacks in the field. We identify that the optimal simulatable, deterministic defense is one with supersequences computed over the correct anonymity sets. We show how to construct a class of such defenses based on how much information the defender is expected to have, and we evaluate these defenses based on approximations over supersequence computation and anonymity set selection. We show a significantly improved overhead over previous simulatable, deterministic defenses such as BuFLO and Tamaraw at the same security level.

Acknowledgements We would like to thank the anonymous reviewers for their suggestions. This research was funded by NSERC, ORF, and The Tor Project, Inc. This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: `www.sharcnet.ca`) and Compute/Calcul Canada.

References

[1] Alexa — The Web Information Company. www.alexa.com.

[2] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Privacy Enhancing Technologies*, pages 1–11. Springer, 2006.

[3] X. Cai, R. Nithyanand, and R. Johnson. New Approaches to Website Fingerprinting Defenses. *arXiv*, abs/1401.6022, 2014.

[4] X. Cai, X. Zhang, B. Joshi, and R. Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 605–616, 2012.

[5] H. Cheng and R. Avnur. Traffic Analysis of SSL-Encrypted Web Browsing. `http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps`.

[6] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 332–346, 2012.

[7] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies*, pages 239–258, 2012.

[8] Y. Gilad and A. Herzberg. Spying in the Dark: TCP and Tor Traffic Analysis. In *Privacy Enhancing Technologies*, pages 100–119. Springer, 2012.

[9] X. Gong, N. Kiyavash, and N. Borisov. Fingerprinting Websites using Remote Traffic Analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 684–686. ACM, 2010.

[10] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, pages 31–42, 2009.

[11] A. Hintz. Fingerprinting Websites Using Traffic Analysis. In *Privacy Enhancing Technologies*, pages 171–178. Springer, 2003.

[12] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2012.

[13] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[14] M. Liberatore and B. Levine. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 255–263, 2006.

[15] L. Lu, E.-C. Chang, and M. C. Chan. Website Fingerprinting and Identification Using Ordered Feature Sequences. In *Computer Security–ESORICS 2010*, pages 199–214. Springer, 2010.

[16] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Proceedings of the 18th Network and Distributed Security Symposium*, 2011.

[17] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th ACM Workshop on Privacy in the Electronic Society*, pages 103–114, 2011.

[18] M. Perry. Experimental Defense for Website Traffic Fingerprinting. https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting, September 2011. Accessed Feb. 2014.

[19] M. Perry. A Critique of Website Traffic Fingerprinting Attacks. https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks, November 2013. Accessed Feb. 2014.

[20] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical Identification of Encrypted Web Browsing Traffic. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 19–30. IEEE, 2002.

[21] Tor. Tor Metrics Portal. https://metrics.torproject.org/. Accessed Oct. 2013.

[22] T. Wang and I. Goldberg. Comparing website fingerprinting attacks and defenses. Technical Report 2013-30, CACR, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-30.pdf.

[23] T. Wang and I. Goldberg. Improved Website Fingerprinting on Tor. In *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society*, 2013.

[24] C. Wright, S. Coull, and F. Monrose. Traffic Morphing: An Efficient Defense against Statistical Traffic Analysis. In *Proceedings of the 16th Network and Distributed Security Symposium*, pages 237–250, 2009.

TapDance: End-to-Middle Anticensorship without Flow Blocking

Eric Wustrow
University of Michigan
ewust@umich.edu

Colleen M. Swanson
University of Michigan
cmswnsn@umich.edu

J. Alex Halderman
University of Michigan
jhalderm@umich.edu

Abstract

In response to increasingly sophisticated state-sponsored Internet censorship, recent work has proposed a new approach to censorship resistance: end-to-middle proxying. This concept, developed in systems such as Telex, Decoy Routing, and Cirripede, moves anticensorship technology into the core of the network, at large ISPs outside the censoring country. In this paper, we focus on two technical obstacles to the deployment of certain end-to-middle schemes: the need to selectively block flows and the need to observe both directions of a connection. We propose a new construction, TapDance, that removes these requirements. TapDance employs a novel TCP-level technique that allows the anticensorship station at an ISP to function as a passive network tap, without an inline blocking component. We also apply a novel steganographic encoding to embed control messages in TLS ciphertext, allowing us to operate on HTTPS connections even under asymmetric routing. We implement and evaluate a TapDance prototype that demonstrates how the system could function with minimal impact on an ISP's network operations.

1 Introduction

Repressive governments have deployed increasingly sophisticated technology to block disfavored Internet content [5, 50]. To circumvent such censorship, many users employ systems based on encrypted tunnels and proxies, such as VPNs, open HTTPS proxies, and a variety of purpose-built anticensorship tools [1, 2, 13, 25, 37]. However, censors are able to block many of these systems by discovering and banning the IP addresses of the servers on which they rely [46, 47]. Some services attempt to remain unblocked by frequently changing their IP addresses, but they face a tension between the desire to make their network locations known to would-be users and the need to keep the same information secret from the censor.

To avoid this tension and escape the cat-and-mouse game that results between censors and anticensorship tools, researchers have recently introduced a new approach called *end-to-middle* (E2M) *proxying* [21, 26, 49]. In an E2M system, friendly network operators agree to help users in other, censored countries access blocked information. These censored users direct traffic toward uncensored "decoy" sites, but include with such traffic a special signal (undetectable by censors) through which they request access to different, censored destinations. Participating friendly networks, upon detecting this signal, redirect the user's traffic to the censored destination. From the perspective of the censor — or anyone else positioned between the censored user and the friendly network — the user appears to be in contact only with the decoy site. In order to block the system, the censor would have to block all connections that pass through participating ISPs, which would result in a prohibitive level of overblocking if E2M systems were widely deployed at major carriers.

Deployment challenges While E2M approaches appear promising compared to traditional proxies, they face technical hurdles that have thus far prevented any of them from being deployed at an ISP. All existing schemes assume that participating ISPs will be able to selectively block connections between users and decoy sites. Unfortunately, this requires introducing new hardware in-line with backbone links, which adds latency and introduces a possible point of failure. ISPs typically have service level agreements (SLAs) with their customers and peers that govern performance and reliability, and adding in-line flow-blocking components may violate their contractual obligations. Additionally, adding such hardware increases the number of components to check when a failure does occur, even in unrelated parts of the ISP's network, potentially complicating the investigation of outages and increasing downtime. Given these risks, ISPs are reluctant to add in-line elements to their networks. In private discussions with ISPs, we found that despite being willing to assist Internet freedom in a technical and even financial capacity, none were willing to deploy existing E2M technologies due to these potential operational impacts.

Furthermore, our original E2M proposal, Telex, assumes that the ISP sees traffic in both directions, client-decoy and decoy-client. While this might be true when the ISP is immediately upstream from the decoy server, it does not generally hold farther away. IP flows are often asymmetric, such that the route taken from source to destination may be different from the reverse path. This asymmetry limits an ISP to observing only one side of a connection. The amount of asymmetry is ISP-dependent, but tier-2 ISPs typically see lower amounts of asymmetry (around 25% of packets) than tier-1s, where up to 90% of packets can be part of asymmetric flows [48]. This severely constrains where in the network E2M schemes that require symmetric flows can be deployed.

Our approach In this paper, we propose TapDance, a novel end-to-middle proxy approach that removes these obstacles to deployment at the cost of a moderate increase in its susceptibility to active attacks by the censor. TapDance is the first E2M proxy that works without an inline-blocking or redirecting element at an ISP. Instead, our design requires only a passive tap that observes traffic transiting the ISP and the ability to inject new packets. TapDance also includes a novel connection tagging mechanism that embeds steganographic tags into the ciphertext of a TLS connection. We make use of this to allow the system to support asymmetric flows and to efficiently include large steganographic payloads in a single packet.

Although TapDance appears to be more feasible to deploy than previous E2M designs, this comes with certain tradeoffs. As we discuss in Section 5, there are several active attacks that a censor could perform on live flows in order to distinguish TapDance connections from normal traffic. We note that each of the previous E2M schemes is also vulnerable to at least some active attacks. As a potential countermeasure, we introduce *active defense* mechanisms, which utilize E2M's privileged vantage point in the network to induce false positives for the attacker.

Even with these tradeoffs, TapDance provides a realistic path to deployment for E2M proxy systems. Given the choice between previous schemes that appear not to be practically fieldable and our proposal, which better satisfies the constraints of real ISPs but requires a careful defense strategy, we believe TapDance is the more viable route to building anticensorship into the Internet's core.

Organization Section 2 reviews the three existing E2M proposals. Section 3 introduces our chosen ciphertext steganography mechanism, and Section 4 explains the rest of the TapDance construction. In Section 5, we analyze the security of our scheme and propose active defense strategies. In Section 6, we compare TapDance to previous E2M designs. We describe our proof-of-concept implementation in Section 7 and evaluate its performance in Section 8. We discuss future work in Section 9 and related work in Section 10, and we conclude in Section 11.

2 Review of Existing E2M Protocols

There are three original publications on end-to-middle proxying: Telex [49], Decoy Routing [26], and Cirripede [21]. The designs for these three systems are largely similar, although some notable differences exist. Figure 1 show the Telex scheme, as one example.

In each design, a client wishes to reach a censored website. To do so, the client creates an encrypted connection to an unblocked *decoy* server, with the connection to this server passing through a cooperating ISP (outside the censored country) that has deployed an *ISP station*. The decoy can be any server and is oblivious to the operation of the anticensorship system. The ISP station determines that a particular client wishes to be proxied by recognizing a *tag*. In Telex, this is a public-key steganographic tag placed in the random nonce of the ClientHello message of a Transport Layer Security (TLS) connection [12]. In Cirripede, users register their IP address with a registration server by making a series of TCP connections, encoding a similar tag in the initial sequence numbers (ISNs). In Decoy Routing, the tag is placed in the TLS client nonce as in Telex, but the client and the ISP station are assumed to have a shared secret established out of band.

In both Telex and Cirripede, the tag consists of an elliptic curve Diffie-Hellman (ECDH) public key point and a hash of the ECDH secret shared with the ISP station. In Decoy Routing, the tag consists of an HMAC of the previously established shared secret key, the current hour, and a per-hour sequence number. In all cases, only the station can observe this tag, using its private key or shared secret.

Once the station has determined that a particular flow should be proxied, all three designs employ an inline blocking component at the ISP to block further communication between the client and the decoy server. Telex and Decoy Routing both block only the tagged flow using an inline-blocking component. Cirripede blocks all connections from a registered client. Cirripede's inline blocking is based on the client's IP address and has a long duration, possibly making it easier to implement than the flow-based blocking used in Telex and Decoy Routing.

After the TLS handshake has completed and the client-server communication is blocked, all three designs have the station impersonate the decoy server, receiving packets to and spoofing packets from its IP address. In Telex, the station uses the tag in the TLS client nonce to compute a shared secret with the client, which the client uses to seed its secret keys during the key exchange with the decoy server. Using this seed and the ability to observe both sides of the TLS handshake, Telex derives the master secret under which the TLS client-server communication is encrypted, and continues to use this shared secret between station and client. In Cirripede and Decoy Routing, the station changes the key stream to be encrypted under

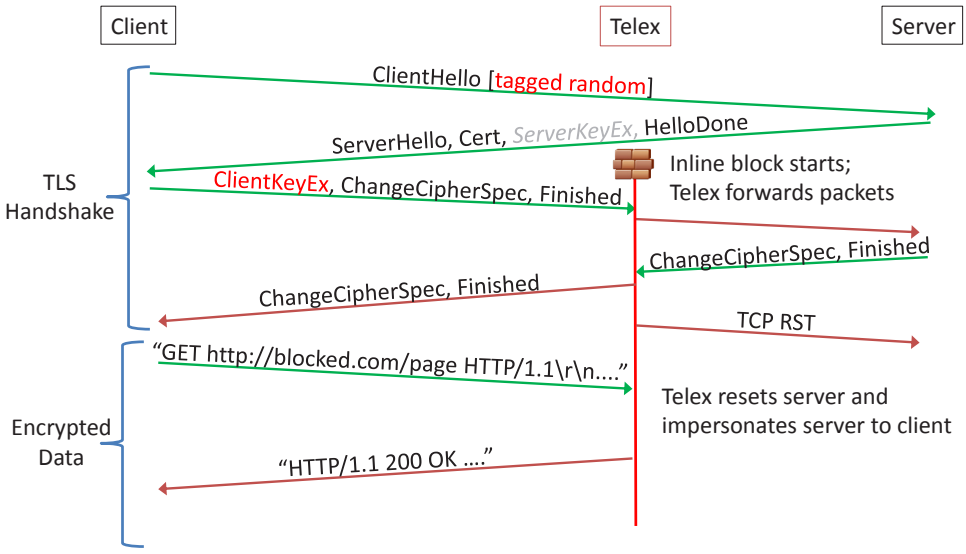


Figure 1: **Telex End-to-Middle Scheme** — To establish communication with an ISP-deployed Telex station, the client performs a TLS connection with an unblocked decoy server. In the TLS ClientHello message, it replaces the random nonce with a public-key steganographic tag that can be observed by the Telex station at the on-path ISP outside the censored country. When the station detects this tag with its private key, it blocks the true connection using an inline-blocking component and forwards packets for the remainder of the handshake. Once the handshake is complete, Telex stops forwarding and begins to spoof packets from the decoy server in order to communicate with the client. While here we only show the details of Telex, all of the first generation ISP proxies (Telex, Cirripede, and Decoy Routing) are similar in architecture; we note differences in Section 2.

the secret exchanged during registration (Cirripede) or previously shared (Decoy Routing).

Changing the communication to a new shared secret opens Cirripede and Decoy Routing to replay and preplay attacks by the adversary. If an adversary suspects a user is accessing these proxies, it can create a new connection that replays parts from the suspected connection and receive confirmation that a particular flow uses the proxy. For example, in Decoy Routing, the adversary can simply use the suspected connection's TLS client nonce in a new connection and send a request. If the first response cannot be decrypted with the client-server shared secret, it confirms that the particular nonce was tagged. For Cirripede, a similar replay of the tagged TCP SYN packets will register the adversary's client, and a connection to the decoy server over TLS will confirm this: if the adversary can decrypt the TLS response with the established master secret, the adversary is not registered with Cirripede, indicating that the TCP SYN packets were not a secret Cirripede tag. Otherwise, if the adversary cannot decrypt the response, this indicates that the SYN packets were indeed a Cirripede tag.

Telex is not vulnerable to either of these attacks, because the client uses the client-station shared secret to seed its half of the key exchange. This allows the station to also compute the client-server shared master secret and verify that the client has knowledge of the client-server shared secret by verifying the TLS finished messages. If an adversary attempted to replay the client random in a new connection, Telex would be able to determine that the user (in this case, the adversary) did not have knowledge of the client-station shared secret, because the user did not originally generate the Diffie-Hellman tag. Thus, Telex is unable to decrypt and verify the TLS finished messages as expected, and will not spoof messages from the server.

Both Cirripede and Decoy Routing function in the presence of asymmetric flows. In Cirripede, the station only needs to observe communication from the client to the decoy server in order to establish its shared secret with the client. In Decoy Routing, the client sends any missing information (i.e., information contained in messages from the server to the client) via another covert channel. In contrast, Telex's approach does not handle asymmetric paths, as the station needs to see both sides of the communication in order to learn the client-server shared master secret.

Unlike any of the existing schemes, TapDance functions without an inline blocking component, potentially making it much easier to deploy at ISPs. Unlike Telex, it supports asymmetric flows, but in doing so it sacrifices some of Telex's resistance to active attacks. We defer a complete comparison between TapDance and the first-generation E2M schemes until Section 6, after we have introduced the details of the system.

3 Ciphertext Covert Channel

Previous E2M covert channels have been limited in size, forcing implementations to use small payloads or several flows in order to steganographically communicate enough information to the ISP station. However, because TapDance does not depend on inline flow-blocking and must work with asymmetric flows, we need a way to communicate the client's request directly to the TapDance station while maintaining a valid TLS session between the client and the decoy server. We therefore introduce a novel technique, *chosen-ciphertext steganography*, which allows us to encode a much higher bandwidth steganographic payload in the ciphertexts of legitimate (i.e., censor-allowed) TLS traffic.

The classic problem in steganography is known as the *prisoners' problem*, formulated by Simmons [41]: two prisoners, Alice and Bob, wish to send hidden messages in the presence of a jailer. These messages are disguised in legitimate, public communication between Alice and Bob in such a way that the jailer cannot detect their presence. Many traditional steganographic techniques focus on embedding hidden messages in non-uniform cover channels such as images or text [4]; in the network setting, each layer of the OSI model may provide potential cover traffic [19] of varying bandwidths. To avoid detection, these channels must not alter the expected distribution of cover traffic [32]. In addition, use of header fields in network protocols for steganographic cover limits the carrying capacity of the covert channel.

We observe it is possible for the sender to use stream ciphers and CBC-mode ciphers as steganographic channels. This allows a sender Alice to embed an arbitrary hidden message to a *third party*, Bob, inside a *valid* ciphertext for Cathy. That is, Bob will be able to extract the hidden message and Cathy will be able to decrypt the ciphertext, without alerting outside entities (or, indeed, Cathy, subject to certain assumptions) to the presence of the steganographic messages.

Moreover, through this technique, we can place limited constraints on the plaintext (such as requiring it be valid base64 or numeric characters), while encoding arbitrary data in the corresponding ciphertext. This allows us to ensure not only that Cathy can decrypt the received ciphertext, but also that the plaintext is consistent with the protocol used. Note that this is a departure from the original prisoners' problem, as we assume Alice is allowed to securely communicate with Cathy, so long as this communication looks legitimate to outside entities.

As our technique works both with stream ciphers and CBC-mode ciphers, which are the two most common modes used in TLS [28], we will use this building block to encode steganographic tags and payloads in the ciphertext of TLS requests.

3.1 Chosen-Ciphertext Steganography

To describe our technique, we start with a stream cipher in counter mode. The key observation is that counter mode ciphers, even with authentication tags, have ciphertexts that are *malleable* from the perspective of the sender, Alice. That is, stream ciphers have the general property of *ciphertext malleability*, in that flipping a single bit in the ciphertext flips a single corresponding bit in the decrypted plaintext. Alice can likewise change bits in the plaintext to effect specific bits in the corresponding ciphertext. Since Alice knows the keystream for the stream cipher, she can choose an arbitrary string that she would like to appear in the ciphertext, and compute (decrypt) the corresponding plaintext. Note that this does not invalidate the MAC or authentication tag used in addition to this cipher, because Alice first computes a valid plaintext, and then encrypts and MACs it using the standard library, resulting in ciphertext that contains her chosen steganographic data.

Furthermore, Alice can "fix" particular bits in the plaintext and allow the remaining bits to be determined by the data encoded in the ciphertext. For example, Alice could require that each plaintext byte starts with 5 bits set to 00110, and allow the remaining 3 bits to be chosen by the ciphertext. In this way, the plaintext will always be an ASCII character from the set "01234567" and the ciphertext has a steganographic "carrying capacity" to encode 3 bits per byte.

While it seems intuitive that Alice can limit plaintext bits for stream ciphers, it may not be as intuitive to see how this is also possible for CBC-mode ciphers. However, while the ciphertext malleability of stream ciphers allows Alice partial control over the resulting plaintext, we show that it is also possible to use this technique in other cipher modes, with equal control over the plaintext values.

In CBC mode, it is possible to choose the value of an arbitrary ciphertext block (e.g., C_2), and decrypt it to compute an intermediary result. This intermediary result must also be the result of the current plaintext block (P_2) xored with the previous ciphertext block (C_1) in order to encrypt to the chosen ciphertext value. This means that, given a ciphertext block, we can choose either the plaintext value (P_2), or the previous ciphertext block (C_1), and compute the other. However, we can also choose a mixture of the two; that is, for each bit we pick in the plaintext, we are "forced" to choose that corresponding bit in the previous plaintext block and vise-versa. Choosing any bits in a ciphertext block (C_1) will force us to repeat this operation for the previous plaintext block (P_1) and twice previous ciphertext block (C_0). We can choose to pick the value of plaintext blocks (fixing the corresponding ciphertext blocks), all the way

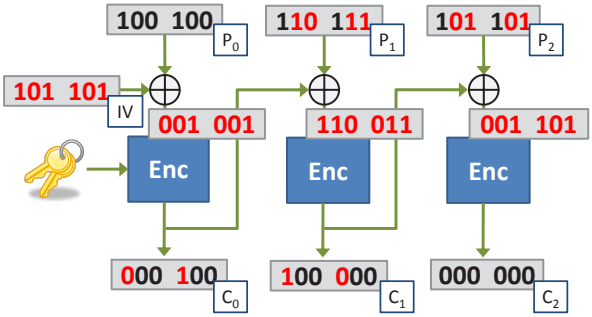


Figure 2: **CBC Chosen Ciphertext Example** — In this example, bits chosen by the encoding are in black, while bits "forced" by computation are red. For example, we choose all 6-bits to be 0 in the last ciphertext block. This forces the block's intermediary to be "forced" to a value beyond our control; in this case 001101. To obtain this value, we can choose a mixture of bits in the plaintext, which forces the corresponding bits in the previous ciphertext block. In this example, we choose the plaintext block to be of the form $1xx1xx$, allowing us to choose 4-bits in the ciphertext, which we choose to be 0s. Thus, the ciphertext has the form $x00x00$. We solve for the unknown bits in the ciphertext and plaintext ($1xx1xx \oplus x00x00 = 001101$) to fill in the missing "fixed" values. We can repeat this process backward until the first block, where we simply compute the IV in order to allow choosing all the bits in the first plaintext block.

back to the first plaintext block, where we are left to decide if we want to choose the value of the first plaintext block or the Initialization Vector (IV) value. At this point, fixing the IV is the natural choice, as this leaves us greater control over the first plaintext block. Figure 2 shows an example of this backpropagation, encoding a total of 4-bits per 6-bit ciphertext block (plus a full final block).

This scheme allows us to restrict plaintexts encrypted with CBC to the same ASCII range as before, while still allowing us to encode arbitrary-length messages in the ciphertext.

While the sender can encode any value in the ciphertext in this manner, we do not wish to change the expected ciphertext distribution. The counter and CBC modes of encryption both satisfy indistinguishability from random bits [38], so encoding anything that is distinguishable from a uniform random string would allow third parties (e.g., a network censor) to detect this covert channel. To prevent this, Alice encrypts her hidden message if necessary, using an encryption scheme that produces ciphertexts indistinguishable from random bits. The resulting ciphertext for Bob is then encoded in the CBC or stream-cipher ciphertext as outlined above. To an outside adversary, this resulting "ciphertext-in-ciphertext" should still be a string indistinguishable from random, as expected.

4 TapDance Architecture

4.1 Protocol Overview

The TapDance protocol requires only a passive network tap and traffic injection capability, and is carefully designed to work even if the station is unable to observe communication between the decoy server and the client. To accomplish this, we utilize several tricks gleaned from a close reading of the TCP specification [35] to allow the TapDance station to impersonate the decoy server without blocking traffic between client and server.

Figure 3 gives an overview of the TapDance protocol. In the first step, the client establishes a normal TLS connection to the decoy web server. Once this handshake is complete, the client and decoy server share a master secret, which they use to generate encryption keys, MAC keys, and initialization vector or sequence state.

The TapDance protocol requires the client to leak knowledge of the client-server master secret, thereby allowing the station to use this shared secret to encrypt all communications. The client encodes the master secret as part of a steganographic tag visible only to the TapDance station. This tag is hidden in an *incomplete* HTTP request sent to the decoy server through the encrypted channel. Since this request is incomplete, the decoy server will not respond with data to the client; this can be accomplished, for example, by simply withholding the two consecutive line breaks that mark the end of an HTTP request. The decoy server will acknowledge this data only at the TCP level by sending a TCP ACK packet and will then wait for the rest of the client's incomplete HTTP request until it times out. As shown in Figure 5, our evaluation reveals that most TLS hosts on the Internet will leave such incomplete request connections open for at least 60 seconds before sending additional data or closing the connection.

When the TapDance station observes this encrypted HTTP request, it is able to extract the tag (and hence the master secret), as discussed in detail in Section 4.2. The station then spoofs an encrypted response from the decoy server to the client. This message acts as confirmation for the client that the TapDance station is present. In particular, this message is consistent with a pipelined HTTPS connection, so by itself does not indicate that TapDance is in use.

At the TCP level, the client acknowledges this spoofed data with a TCP ACK packet, and because there is no inline-blocking between it and the server, the ACK will reach the server. However, because the acknowledgment number is above the server's $SND.NXT$, the server will not respond. Similarly, if the client responds with additional data, the acknowledgment field contained in those TCP packets will also be beyond what the server has sent. This allows the TapDance station to continue to imper-

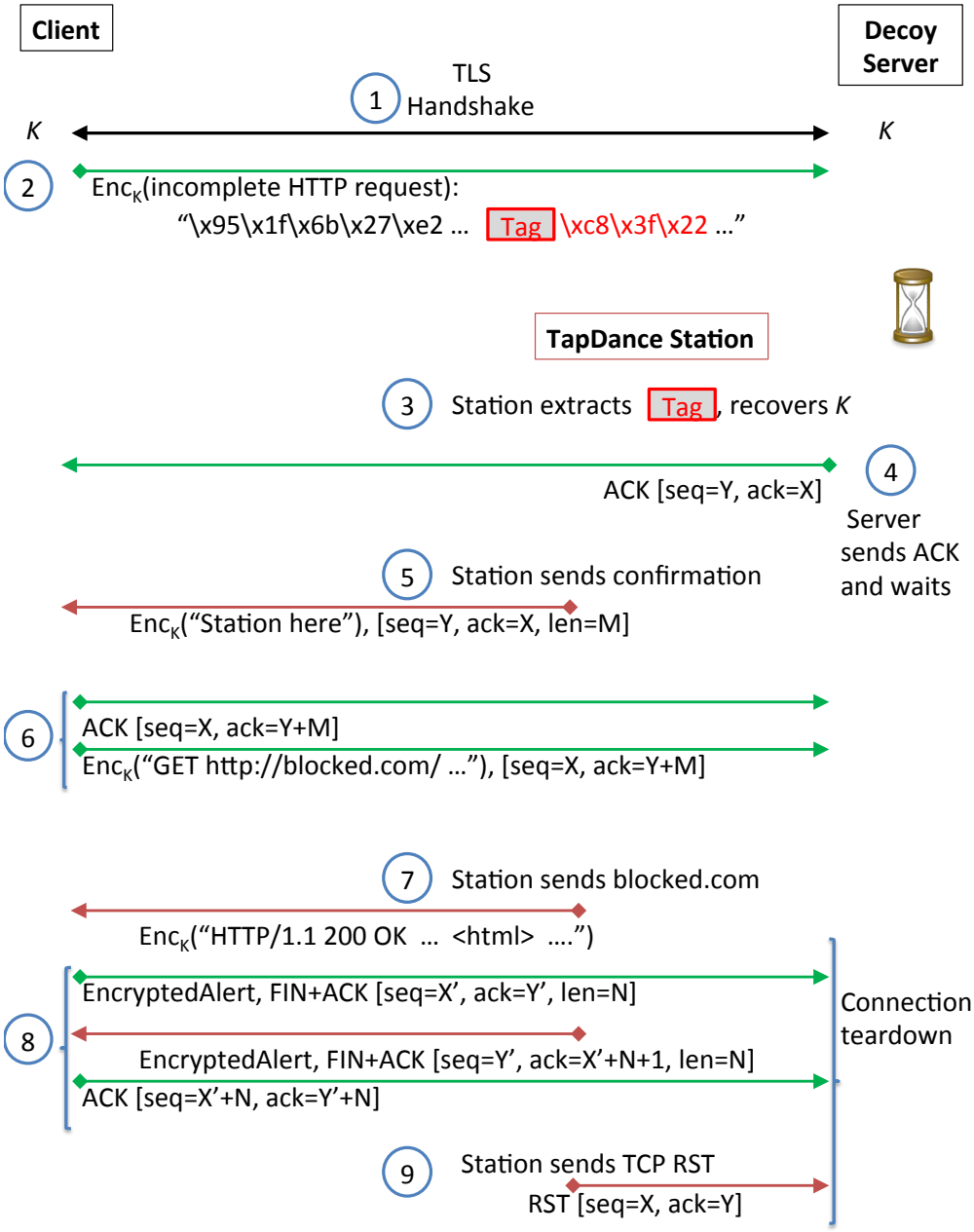


Figure 3: **TapDance Overview** — (1) The client performs a normal TLS handshake with an unblocked decoy server, establishing a session key K. (2) The client sends an *incomplete* HTTP request through the connection and encodes a steganographic tag in the *ciphertext* of the request, using a novel encoding scheme (Section 4.2). (3) The TapDance station observes and extracts the client's tag, and recovers the client-server session secret K. (4) The server sends a TCP ACK message in response to the incomplete HTTP request and waits for the request to be completed or until it times out. (5) The station, meanwhile, spoofs a response to the client from the decoy server. This message is encrypted under K and indicates the station's presence to the client. (6) The client sends a TCP ACK (for the spoofed data) and its real request (blocked.com). The server ignores both of these, because the TCP acknowledgment field is higher than the server's TCP SND.NXT. (7) The TapDance station sends back the requested page (blocked.com) as a spoofed response from the server. (8) When finished, the client and TapDance station simulate a standard TCP/TLS authenticated shutdown, which is again ignored by the true server. (9) After the connection is terminated by the client, the TapDance station sends a TCP RST packet that is valid for the server's SND.NXT, silently closing its end of the connection before its timeout expires.

sonate the server, acknowledging data the client sends, and sending its own data in response, without interference from the server itself.

4.2 Tag Format

In TapDance, we rely on elliptic curve Diffie-Hellman to agree on a per-connection shared secret between the client and station, which is used to encrypt the steganographic tag payload. The tag consists of the client's connection-specific elliptic curve public key point ($Q = eG$), encoded as a string indistinguishable from uniform, followed by a variable-length encrypted payload used to communicate the client-server TLS master secret (and intent for proxying) to the station.

In order to properly disguise the client's elliptic curve point, we use Elligator 2 [8] over Curve25519 [7]. Elligator 2 is an efficient encoding function that transforms, for certain types of elliptic curves, exactly half of the points on the curve to strings that are indistinguishable from uniform random strings.

The client uses the TapDance station's public key point ($P = dG$) and its own private key (e) to compute an ECDH shared secret with the station ($S = eP = dQ$), which is used to derive the payload encryption key. The encrypted payload contains an 8-byte magic value used by the station to detect successful decryption, the client and server random nonces, and the client-server master secret of the TLS connection. With this payload, typically contained in a single packet from the client, the station is able to derive the TLS master secret between client and server.

We insert the tag, composed of the encoded point and encrypted payload, into the ciphertext of the client's incomplete request to the server using the chosen ciphertext steganographic channel described in Section 3. In order to avoid the server generating unwanted error messages, we maintain some control over the plaintext that the server receives using the plaintext-limiting technique as described in Section 3. Specifically, we split the tag into 6-bit chunks and encode each chunk in the low order bits of a ciphertext byte. This allows the two most significant bits to be chosen freely in the plaintext (i.e. not decided by the decryption of the tag-containing ciphertext). We choose these two bits so that the plaintext always falls within the ASCII range 0x40 to 0x7f. We verified that Apache was capable of handling this range of characters in a header line without triggering an error.

5 Security Analysis

Our threat model is similar to that of previous end-to-middle designs. We assume an adversarial censor that can observe, alter, block, or inject network traffic within their domain or geographic region (i.e., country) and may gain access to foreign resources, such as VPNs or private servers, by leasing them from providers. Despite control over its network infrastructure, however, we assume the censor does not have control over end-users' computers, such as the ability to install arbitrary programs or Trojans.

The censor can block its citizens' access to websites it finds objectionable, proxies, or other communication endpoints it chooses, using IP blocking, DNS blacklists, and deep-packet inspection. We assume the censor uses blacklisting to block resources and that the censor does not wish to block legitimate websites or otherwise cut themselves off from the rest of the Internet, which may inhibit desirable commerce or communication. In addition, we assume that the censor allows end-to-end encrypted communication, specifically TLS communication. As websites increasingly support HTTPS, censors face increasing pressures against preventing TLS connections [14].

While the threat model for TapDance is similar to those assumed by prior end-to-middle schemes, our fundamentally new design has a different attack surface than the others. We perform a security analysis of TapDance and compare it to the previous generation designs, focusing on the adversarial goal of distinguishing normal TLS connections from TapDance connections. In particular, we do not attempt to hide the deployment locations of the TapDance stations themselves.

5.1 Passive Attacks

TLS handshake TLS allows implementations to support many different extensions and cipher suites. As a result, implementations can be easy to differentiate based on the ciphers and extensions they claim to support in their ClientHello or ServerHello messages. In order to prevent this from being used to locate suspicious implementations, our proxy must blend in to or mimic another popular client TLS implementation. For example, we could support the same set of ciphers and extensions as Chrome for the user's platform. Currently, our client mimics Chrome's cipher suite list for Linux.

Cryptographic attacks A computationally powerful adversary could attempt to derive the station's private key from the public key. However, our use of ECC Curve25519 should resist even the most powerful computation attacks using known discrete logarithm algorithms. The largest publicly known ECC key to be broken is only 112 bits, broken over 6 months in 2009 on a 200-PlayStation3 cluster [9]. In contrast to Telex, TapDance also supports increasing the key size as needed, as we are not limited to a fixed field size for our tag.

Forward secrecy An adversary who compromises an ISP station or otherwise obtains a station's private key can use it to trivially detect both future and previously

recorded flows in order to tell if they were proxy flows. Additionally, they can use the key to decrypt the user's request (and proxy's response), learning the censored websites users have visited. To address the first problem, we can use a technique suggested in Telex [49]. The ISP station generates many private keys ahead of time and stores them in either a hardware security module or offline storage, and provides all of the public keys to the clients. Clients can then cycle through the public keys they use based on a course-grained time (e.g., hours or days). The proxy could also cycle through keys, destroying expired keys and limiting access to future ones.

To address the second problem, TapDance is compatible with existing forward-secure protocols. For example, for each new connection it receives, the TapDance station can generate a new ECDH point randomly, and establish a new shared secret between this new point and the original point sent by the client in the connection tag. The station sends its new ECDH public point to the client in its Hello message, and the remainder of the connection is encrypted under the new shared secret. This scheme has the advantage that it adds no new round trips to the scheme and only 32-bytes to the original ISP station's response.

Packet timing and length The censor could passively measure the normal round-trip time between potential servers and observe the set of packet lengths of encrypted data that a website typically returns. During a proxy connection, the round-trip time or the packet lengths of the apparent server may change for an observant censor, as the station may be closer or have more processing delay than the true server. This attack is possible on all three of the first generation E2M schemes, as detailed in [40]. However, such an attack at the application level may be difficult to carry out in practice, as larger, legitimate websites may have many member-only pages that contain different payload lengths and different processing overhead. The censor must be able to distinguish between "blind pages" it cannot confirm are part of the legitimate site and decoy proxy connections. We note that this is difficult at the application level, but TCP round-trip times may have a more consistent and distinguishable difference.

Lack of server response If the TapDance station fails to detect a client's flow, it will not respond to the client. This may appear suspicious to a censor, as the client sends a request, but there is no response at the application level from the server. This scenario could occur for three reasons. First, the censor may disrupt the path between client and TapDance station in order to cause such a timeout, using one of the active attacks below (such as the routing-around attack), in order to confirm a particular flow is attempting to use TapDance. Second, such false pickups may happen intermittently (due to ISP station malfunction). Finally, a client may attempt to find new TapDance stations by probing many potential decoy servers with tagged TLS connections. Paths that do not contain ISP stations will have suspiciously long server response times.

To address the last issue, probing clients could send complete requests and tag their requests with a secret nonce. The station could record these secret nonces, and, at a later time (out of band, or through a different TapDance station), the client can query the station for the secret nonces it sent. In this way, the client learns new servers for which the ISP station is willing to proxy without revealing the probing pattern. To address the first two problems, we could have clients commonly use servers that support long-polling HTTP push notification. In these services, normal requests can go unanswered at the application layer as long as the server does not have data to send to the client, such as in online-gaming or XMPP servers. Another defense is to have the client send complete requests that force the server to keep the connection alive for additional requests, and to have the TapDance station inject additional data *after* the server's initial response. This requires careful knowledge of the timing and length of the server's initial response, which could either be provided by active probing from the station or information given by the client.

TCP/IP protocol fingerprinting The adversary could attempt to observe packets coming from potential decoy servers and build profiles for each server, including the set of TCP options supported by the server, IP TTL values, TCP window sizes, and TCP timestamp slope and offset. If these values ever change, particularly in the middle of a connection (and only for that connection), it could be a strong indication of a particular flow using a proxy at an on-path ISP. To prevent this attack, the station also needs to build these profiles for servers, either by actively collecting this profile from potential servers, or passively observing the server's responses to non-proxy connections and extracting the parameters. Alternatively, the client can signal to the station some of the parameters. First generation schemes varied in defense for this type of attack; for example, Telex's implementation is able to infer and mimic all of these parameters from observing the servers' responses, although Telex requires a symmetric path in order to accomplish this. In theory, parameters that the adversary can measure for fingerprinting can also be measured by the station and mimicked. However, given that the adversary has only to find one distinguisher in order to succeed, server mimicry remains difficult to achieve in practice.

5.2 Active Attacks

TLS attacks The censor may issue fake TLS certificates from a certificate authority under its control and then target TLS sessions with a man-in-the-middle attack.

While TapDance and previous designs are vulnerable to this attack, there may be external political pressure that discourages a censor from this attack, as it may be disruptive to foreign e-commerce in particular. We also argue that as the number of sites using TLS continues to increase, this attack becomes more expensive for the censor to perform without impacting performance. Finally, decoy servers that use certificate pinning or other CA-protection mechanisms such as Perspectives [45], CAge [27], or CA country pinning [42], can potentially avoid such attacks.

Packet injection Because TapDance does not block packets from the client to the true server, it is possible for the censor to inject spoofed probes from the client that will reach the server. If the censor can craft a probe that will result in the server generating a response that reveals the server's true TCP state, the censor will be able to use this response to differentiate real connections from proxy connections. While the previous designs also faced this threat [40], the censor had to inject the spoofed packet in a way that bypassed the station's ISP inline blocking element. In TapDance, there is no blocking element, and so the censor is able to simply send it without any routing tricks. An example of this attack is the censor sending a TCP ACK packet with a stale sequence number, or one for data outside the server's receive window. The server will respond to this packet with an ACK containing the server's TCP state (sequence and acknowledgment), which will be smaller than the last sequence and/or acknowledgments sent by the station.

There are a few ways to deal with this attack if the censor employs it. First, we can simply limit each proxy connection to a single request from the client and a response from the station, followed immediately by a connection close. This will dramatically increase the overhead of the system but will remove the potential for the adversary to use injected packets and their responses to differentiate between normal and proxy connections. This is because the TCP state between the station and real server will not diverge until the station has sent its response, leaving only a very small window where the censor can probe the real server for its state and get a different response.

Active defense Alternatively, in order to frustrate the censor from performing packet injection attacks, we can perform *active defense*, where the station observes active probes such as the TCP ACK and responds to them in a way that would "reveal" a proxy connection, even for flows that are not proxy connections. To the censor, this would make even legitimate non-proxy connections to the server appear as if they were proxy connections.

As an example, consider a censor that injects a stale ACK for suspected proxy connections. Connections that are actually proxy connections will respond with a stale ACK from the server, revealing the connection to the censor. However, the station could detect the original probe, and if it is not a proxied connection, respond with a stale ACK so as to make it appear to the censor as if it were. In this way, for every probe the censor makes, they will detect, sometimes incorrectly, that the connection was a proxy connection.

Replay attacks The censor could capture suspected tags and attempt to replay them in a new connection, to determine if the station responds to the tag. To specifically attack TapDance, the adversary could replay the client's tag-containing request packet after the connection has closed and observe if the station appears to send back a response. We note that both Cirripede and Decoy Routing are also vulnerable to tag replay attacks, although Telex provides some limited protection from them. To protect against duplicated tags, the station could record previous tags and refuse to respond to a repeated tag. To avoid saving all tags, the station could require clients to include a recent timestamp in the encrypted payload[1].

However, such a defense may enable a denial of service attack: the censor could delay the true request of a suspected client and send it in a different connection first. In this *preplay* version of the attack, the censor is also able to observe whether the station responds with the ClientHello message. If it does, the censor will know the suspected request contained a tag.

Denial of service The censor could attempt to exhaust the station's resources by creating many proxy connections, or by sending a large volume of traffic that the ISP station will have to check for tags using an expensive ECC function. We estimate that a single ISP station deployment of our implementation on a 16-core machine could be overwhelmed if an attacker sends approximately 1.2 Gbps of pure TLS application data packets past it. This type of attack is feasible for an attack with a small botnet, or even a few well-connected servers. Because ISPs commonly perform load balancing by flow-based hashing, we can scale our deployment linearly to multiple branches of machines and use standard intrusion detection techniques to ignore packets that do not belong to valid connections or that come from spoofed or blacklisted sources [34].

Routing around the proxy A recent paper by Schuchard et al. details a novel attack against our and previous designs [40]. In this attack, the censor is able to change local routing policy in a way that redirects outbound flows around potential station-deploying ISPs while still allowing them to reach their destinations. This prevents the ISP station from being able to observe the tagged flows and thus from being a proxy for the clients. However, Houmansadr et al. investigate the cost to the

[1] The client random which is sent in the encrypted payload already contains a timestamp for the first 4 bytes

censor of performing such an attack and find it to be prohibitively expensive [23]. Although both of these papers ultimately contribute to deciding which ISPs should deploy proxies in order to be most resilient, we consider such a discussion outside our current scope.

Tunneling around the proxy A more conceptually simple attack is for the censor to transparently tunnel specific suspected flows around the ISP station. For example, the censor could rent a VPN or VPS outside the country and send specific flows through them to avoid their paths crossing the ISP station. This attack is expensive for the adversary to perform, and so could not reasonably be performed for an entire country. However, it could be performed for particular targets and combined with previous passive detection attacks to aid the censor in confirming whether particular users are tagging their flows.

Complicit servers A censor may be able to compromise, coerce, or host websites that can act as servers for decoy connections. The vantage point from a server allows them to observe incomplete requests from clients, including the plaintext that the client mangled in order to produce the tag in the ciphertext. This allows the censor to both observe specific clients using the ISP station and also disrupt use of the proxy with the particular server. There is little TapDance or previous designs can do to avoid cooperation between servers and the censor, as the two can simply compare traffic received and detect proxy flows as ones that have different data at the two vantage points. However, using this vantage point to disrupt proxy use could be detected by clients and the server avoided (and potentially notified in the case of a compromise).

6 Comparison

On the protocol level, TapDance bears more similarity to Telex than Cirripede, in that clients participate in TapDance on a per-connection basis, rather than participating in a separate registration phase as in Cirripede, and in that client-station communication, after the initial Diffie-Hellman handshake, is secured using the client-server master secret. In order to conserve bandwidth, our design, like both Telex and Cirripede, leverages elliptic curve cryptography to signal intent to use the system and to establish a shared secret between client and station.

However, TapDance exhibits several important differences from previous protocols, which has implications for both security and functionality. As discussed in Section 1, one of the largest challenges to deploying E2M proxies at ISPs is the inline flow-blocking component. TapDance has the singular advantage in that it allows client-server communication to continue unimpeded. In fact, our design requires only that the TapDance station be able to passively observe communication from client to server and be able to inject messages into the network; the station can be oblivious to communication passing from server to client.

The advantages of the TapDance protocol stem from its careful use of chosen-ciphertext steganography (described in Section 3) to hide the client's tag and the fact that a high percentage of servers ignore stale TCP-level messages. In contrast, previous proposals rely on inline blocking to prevent server-client communication, and TCP sequence numbers and TLS ClientHello random nonces to disguise the client's steganographic tag. In general, these fields are useful in steganography because these strings should be uniformly random for legitimate connections, providing a good cover for the tag that replaces them, so long as this tag is indistinguishable from random.

However, both of these fields are fixed size; each TLS nonce can be replaced with a 224-bit uniform random tag, and each TCP sequence number with only 24 bits of a tag. Cirripede, which encodes the client's tag into TCP sequence numbers, uses multiple TCP connections to convey the full tag to the station. Telex and Decoy Routing both use a single TLS nonce to encode the client's tag. Given the limited bandwidth of these covert channels, they are useful to convey only short secrets, while the rest of the payload (such as the request for a blocked website) must take place in a future packet.

TapDance, on the other hand, leverages chosen-ciphertext steganography in order to encode steganographic tags in the ciphertext of a TLS connection, without invalidating the TLS session itself. Encoding the tag in the ciphertext has several advantages. First, the tag is no longer constrained to a fixed field size of either 24 or 224 bits, allowing us to encode more information in each tag, and use larger and more secure elliptic curves. Second, because the ciphertext is sent after the TLS handshake has completed, it is possible to encode the connection's master secret in this tag, allowing the station to decrypt the TLS session from a single packet, and without requiring the station to observe packets from the server.

In addition, TapDance takes advantage of recent work by Bernstein et al. [8], in order to disguise elliptic curve points as strings indistinguishable from uniform, namely Elligator 2. Traditional encoding of elliptic curve points is distinguishable from random for several reasons, which are outlined in detail in [8]. Telex and Cirripede address this concern by employing two closely related elliptic curves, which is less efficient than TapDance's use of Elligator 2, as the latter method requires only a single elliptic curve to achieve the same functionality.

From a security perspective, the only attacks unique to TapDance are the lack of server response and packet injection attacks. Besides these, we find our design has no additional vulnerabilities from which all previous designs were immune. While these two attacks do pose a

	Telex [49]	Cirripede [21]	Decoy Routing [26]	TapDance
Steganographic channel	TLS client random	TCP ISNs	TLS client random	TLS ciphertext
Works without inline components	○	○	○	●
Handles asymmetric flows	○	●	●	●
Proxies per flow	●	○	●	●
Replay/preplay attack resistant	●	○	○	○
Traffic analysis defense	○	○	○	○

Table 1: **Comparing E2M Schemes** — Unlike previous work, TapDance operates without an inline flow-blocking component at cooperating ISPs. However, it is vulnerable to active attacks that some previous designs resist. No E2M system yet defends against traffic analysis or website fingerprinting, making this an important area for further study.

threat to TapDance, the benefits of a practical ISP station deployment — at least as a bridge to stronger future systems — may outweigh the potential risks.

In summary, our approach obviates the need for an inline blocking element at the ISP, which is a requirement of Telex, Cirripede, and Decoy Routing, while preserving system functionality in the presence of asymmetric flows, which is an advantage over Telex. In addition, the covert channel used in TapDance is higher bandwidth than that of previous proposals and holds potential for future improvements (e.g., in terms of number of communication rounds required and flexible security levels) of client-station protocols.

7 Implementation

We have implemented TapDance in two parts: a client that acts as a local HTTP proxy for a user's browser, and a station that observes a packet tap at an ISP and injects traffic when it detects tagged connections. Our station code is written in approximately 1,300 lines of C, using libevent, OpenSSL, PF_RING [33], and forge_socket[2].

7.1 Client Implementation

Our client is written in approximately 1,000 lines of C using libevent [29] and OpenSSL [36]. The client currently takes the domain name of the decoy server as a command line argument, and for each new local connection from the browser, creates a TLS connection to the decoy server. Once the handshake completes, the client sends the incomplete response to prevent the server from sending additional data, and to encode the secret tag in the ciphertext as specified in Section 4.2. The request is simply an HTTP request with a valid HTTP request line, "Host" header, and an "X-Ignore" header that precedes the "garbage" plaintext that will be computed to result in the chosen tag appearing in the ciphertext. We have implemented our ciphertext encoding for AES_128_GCM [39], although it also works without modification for AES_256_GCM cipher suites. We have implemented Elligator 2 to work with Curve25519, in order to encode the client's public point in the ciphertext as a string that is indistinguishable from uniform random. After this 32-byte encoded point, the client places a 144-byte encrypted payload. This payload is encrypted using a SHA-256 hash of the 32-byte shared secret (derived from the client's secret and station's public point) using AES-128 in CBC mode. We use the first 16-bytes of the shared secret hash as the key, and the last 16 bytes as the initialization vector (IV). The payload contains an 8-byte magic value, the 48-byte TLS master secret, 32-byte client random, 32-byte server random, and a 16-byte randomized connection ID that allows a client to reconnect to a previous proxy connection in case the underlying decoy connection is prematurely closed.

[2] https://github.com/ewust/forge_socket/

7.2 Station Implementation

Our TapDance station consists of a 16-core Supermicro server connected over a gigabit Ethernet to a mirror port on an HP 6600-24G-4XG switch in front of a well-used Tor exit node generating about 160 Mbps of traffic. The station uses PF_RING, a fast packet capture Linux kernel module, to read packets from the mirror interface. In addition to decreasing packet capture overhead, PF_RING supports flow clustering, allowing our implementation to spread TCP flow capture across multiple processes. Using this library, our station can have several processes on separate cores share the aggregate load.

For each unique flow (4-tuple), we keep a small amount of state whether we have seen an Application Data packet for the flow yet. If we have not, we verify the current packet's TCP checksum, and inspect the packet to determine if it is an Application Data packet. If it is, we mark this flow as having been inspected, and pass the packet ciphertext to the tag extractor function. This function extracts the potential tag from the ciphertext, decoding the client's public point using Elligator 2, generating the shared secret using Curve25519, and hashing it to get the AES decryption key for the payload. The extractor

decrypts the 144-byte payload included by the client, and verifies that the first 8 bytes are the expected magic value. If it is, the station knows this is a tagged flow, and uses the master secret and nonces extracted from the encrypted payload to compute the key block, which contains encryption and decryption keys, sequence numbers or IVs, and MAC keys (if not using authenticated encryption) for the TLS session between the client and server.

This "ciphertext-in-ciphertext" is indistinguishable from random to everyone except the client and station. The 144-byte payload is encrypted using a strong symmetric block cipher (AES-128) in CBC mode, whose key is derived from the client-station shared secret. The remainder of the tag is the client's ECDH public point, encoded using Elligator 2 [8] over Curve25519 [7]. The encoded point is indistinguishable from uniform random due to the properties of the Elligator 2 encoding function.

Once the station has determined the connection is a tagged flow, it sets up a socket in the kernel to allow it to spoof packets from and receive packets for the server using the forge_socket kernel module. The station makes this socket non-blocking, and attaches an SSL object initialized with the extracted key block to it. The station then sends a response to the client over this channel, containing a confirmation that the station has picked up, and the number of bytes that the client is allowed to send toward this station before it must create a new connection.

7.3 Connection Limits

Because the server's connection with the client remains open, the server receives packets from the client, including data and acknowledgments for the station's data. The server will initially ignore these messages, however there are two instances where the server will send data. When it does so, the censor would be able to see this anomalous behavior, because the server will send data with stale sequence numbers and different payloads from what the station sent.

The first instance of the server sending data is when the server times out the connection at the application level. For example, web servers can be configured to timeout incomplete requests after a certain time, by using the mod_reqtimeout[3] module in Apache. We found through our development and testing the shortest timeout was 20 seconds, although most servers had much longer timeouts. We measured TLS hosts to determine how long they would take to time out or respond to an incomplete request similar to one used in TapDance. We measured a 1% sample of the IPv4 address space listening on port 443, and the Alexa top million domains using ZMap [15], and found that many servers had timeouts longer than 5 minutes. Figure 5 shows the fraction of server timeouts.

[3]http://httpd.apache.org/docs/2.2/mod/mod_reqtimeout.html

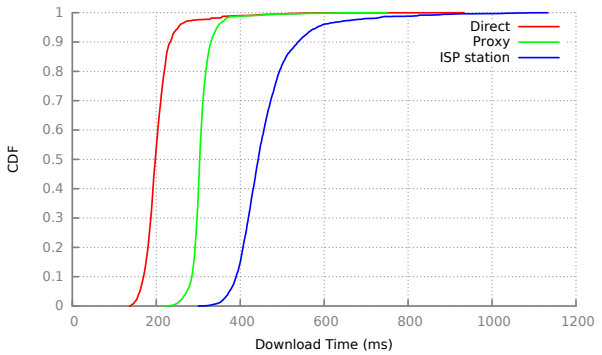


Figure 4: **Download Times Through TapDance** — We used Apache Benchmark to download www.facebook.com 5000 times (with a concurrency of 100) over normal HTTPS, through a single-hop proxy, and through our TapDance proof-of-concept.

The second reason a server will send observable packets back to the client is if the client sends it a sequence number that is outside of the server's current TCP receive window. This happens when the client has sent more than a window's worth of data to the station, at which point the server will respond with a TCP ACK packet containing the server's stale sequence and acknowledgment numbers, alerting an observant censor to the anomaly.

To prevent both of these instances from occurring in our implementation, we limit the connection duration to less than the server's timeout, and we limit the number of bytes that a client can send to the station to up to the server's receive window size. Receive window sizes after the TLS handshake completes are typically above about 16 KB. We note that the station is not limited to the number of bytes it can send to the client per connection, making the 16 KB limit have minimal impact on most download-heavy connections.

In the event that the client wants to maintain its connection for longer than the duration or send more than 16 KB, the client can reuse the 16-byte connection ID in a new E2M TLS connection to the server. The station will decode the connection ID and reconnect the new flow to the old proxy connection seamlessly. This allows the browser to communicate to the HTTP proxy indefinitely, without having to deal with the limitations of the underlying decoy connection.

8 Evaluation

Throughout our evaluation, we used a client running Ubuntu 13.10 connected to a university network over gigabit Ethernet. For our decoy server, we used a Tor exit server at our institution, with a gigabit upstream through an HP 6600-24G-4XG switch. For our ISP station, we used a 16-core Supermicro server with 64 GB of RAM,

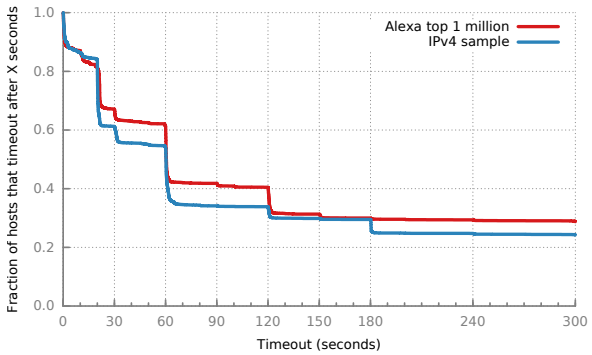


Figure 5: **Timeouts for Decoy Destinations** — To measure how long real TLS hosts will leave a connection open after receiving the incomplete request used in TapDance, we connected to two sets of TLS hosts (the Alexa top 1 million and a 1% sample of the IPv4 address space). We sent TapDance's incomplete request and timed how long the host would leave the connection open before either sending data or closing the connection. We find that over half the hosts will allow connections 60 seconds or longer.

connected via gigabit NICs to an upstream and to a mirror port from the HP switch. Our ISP station is therefore able to observe (but not block) packets to the Tor exit server, which provides a reasonable amount of background traffic on the order of 160 Mbps. In our tests, the Tor exit node generates a modest amount of realistic user traffic. Although not anywhere near the bandwidth of a Tier-1 ISP, Tor exit nodes generate a greater ratio of HTTPS flows than a typical ISP (due to the Tor browser's inclusion of the HTTPS Everywhere plugin), and we can use this microbenchmark to perform a back-of-the-envelope calculation to the loads we would see at a 40 Gbps Transit ISP tap.

We evaluate our proof-of-concept implementation with the goal of demonstrating that our system operates as described, and that our implementation is able to function within the constraints of our mock-ISP. To demonstrate that our system operates as described, we set Firefox to use our client as a proxy, and browsed several websites while capturing packets on the client and the decoy server. We then manually inspected the recorded packets to confirm that there were no additional packets sent by the server that would reveal our connections to be proxied connections. Empirically, we note that we are able to easily browse the Internet through this proxy, for example watching high-definition YouTube videos.

To evaluate the performance of our system, we created 8 proxy processes on our ISP station, using the same PF_RING cluster ID in order to share the load across 8 cores. The background traffic from the Tor exit server does not appear to have a significant impact on the proxy's load: each process handles between 20 and 50 flows at a given time, comprising up to 35 Mbps of TLS traffic. The CPU load during this time was less than 1%.

We used Apache Benchmark[4] in order to issue 5,000 requests through our station proxy, with a concurrency of 100, and compared the performance for fetching a simple page over HTTP and over HTTPS. We also compare fetching the same pages directly from the server and through a single-hop proxy. Figure 4 shows the cumulative distribution function for the total time to download the page. Although there is a modest overhead for end-to-middle proxy connections compared to direct or simple proxies, the overhead is not prohibitive to web browsing habits; users are still able to interact with the page, and pages can be expected to load in a reasonable time period. In particular, our proxy adds a median latency of 270 milliseconds to a page download in our tests when compared with a direct download.

We find that the CPU performance is bottlenecked by our single-threaded client. During our tests, the client consumes 100% CPU on a single core, while each of the 8 processes on the ISP station consume between 4-7% CPU. We also observe that a majority of the download time is spent waiting for the connection handshake to complete with the server. To improve this performance, we could speculatively maintain a connection pool in order to decrease the wait-time between requests. However, care must be taken in order to mimic the same connection pool behaviors that a browser might exhibit.

We also note that although the distribution of download times appear different for ISP station vs. normal connections, this does not necessarily indicate an observable feature for a censor. This is because our download involves a second round trip between client and server before the data reaches the client. The censor would still have to distinguish between this type of connection behavior and innocuous HTTP pipelined connections. It still may be possible for the censor to distinguish, however, as we discussed in Section 5, traffic analysis is an open problem for existing network proxies, and outside the scope of this paper.

Tag creation and processing In order to evaluate the overhead of creating and checking for tags, we timed the creation and evaluation of 10,000 tags. We were able to create over 2,400 tags/second on our client and verify over 12,000 tags/second on a single core of our ISP station. We find that the majority of time (approximately 80%) during tag creation is spent performing the expected three ECC point multiplications (an expected two to generate the client's Elligator-encodable public point and one to generate the shared secret). Similarly, during tag checking, nearly 90% of the computation time is spent on the single ECC point multiplication. Faster ECC implementations

[4] http://httpd.apache.org/docs/2.2/programs/ab.html

(such as tuned-assembly or ASICs) could have a significant impact toward improving the performance of tag verification on the ISP station.

Server support In order to measure how many servers can act as decoy destinations, we probed samples of the IPv4 address space as well as the Alexa top million hosts with tests to indicate support for TapDance. In our first experiment, we tested how long servers would wait to timeout an incomplete request, such as the one used by the client in TapDance. We scanned TLS servers in a 1% sample of the IPv4 address space, as well as the Alexa top million hosts, and sent listening servers a TLS handshake, followed by an incomplete HTTP request containing the full range of characters used in the TapDance client. We timed how long each server waited to either respond or close the connection. Servers that responded immediately do not support the TapDance incomplete request, either because they do not support incomplete requests, or the request contained characters outside the allowed range. Figure 5 shows the results of this experiment. For the 20-second timeout used in our implementation, over 80% of servers supported our incomplete request.

We also measured how servers handled the out-of-sequence TCP packets sent by the TapDance client, including packets acknowledging data not yet sent by the server. Again, we used a 1% sample of the IPv4 address space and the Alexa top million hosts. For each host, we connected to port 80 and sent an incomplete HTTP request, followed by a TCP ACK packet and a data-containing packet, both with acknowledgements set 100 bytes ahead of the true value. We find that the majority of Alexa servers still allow such packets, however, older or embedded systems often respond to our probes, in violation of the TCP specification. We conclude that TapDance clients must carefully select which servers they use as end points, but that there is no shortage of candidates from which to select.

9 Future Work

The long-term goal of end-to-middle proxies is to be implemented and deployed in a way that effectively combats censorship. While we have suggested a design that we believe is more feasible than previous work, more engineering must be done to bring it to maturity.

For example, deploying an end-to-middle proxy such as TapDance at an ISP requires not only scaling up to meet the demands of proxy users, but also of the deploying ISP's non-proxy traffic, which can be on the order of gigabits per second. One potential solution to this problem is to make the ISP component as stateless as possible. Extending TapDance, it may be possible to construct a "single-packet" version of an end-to-middle proxy. In this version the client uses the ciphertext steganographic channel to encode its entire request to the proxy. The proxy needs only detect these packets, fetch the requesting page, and inject a response. Such a design would not need to reconstruct TCP flows or keep state across multiple packets, allowing it to handle higher bandwidths of traffic, at the expense of making active attacks easier to perform by an adversary. Further investigation may discover an optimal balance between these tradeoffs.

Another open research question is where specifically in the network such proxies should be deployed. Previously, "Routing around Decoys" [40] outlined several novel attacks that a censor could perform in order to circumvent many anticensorship deployment strategies. There is ongoing discussion in the literature about the practical costs of these attacks, and practical countermeasures deployments could take to protect against them [11, 23].

As mentioned in Section 5, traffic fingerprinting is a concern for all proxies, and remains an open problem. Previous work has discussed these attacks as they apply to ISP-located proxies [40] and other covert channel proxies [18, 20]. Future work in this direction could provide insight into how to generate or mimic network traffic and protocols.

Finally, there is room to explore more active defense techniques, as outlined in Section 5. As end-to-middle proxies become more prominent, this is likely to become an important problem, as China has already started to employ active attacks in order to detect and censor Tor bridge relays [13, 46, 47]. Collaborating with ISPs will allow us to explore the technical capabilities and policies that would permit active defense against these attacks.

10 Related Work

Other anticensorship schemes Besides end-to-middle proxies, previous anticensorship approaches, including Collage [10] and Message in a Bottle [24], have leveraged using user-generated content on websites to bootstrap communication between censored users and a centrally-operated proxy. However, these designs are not intended to work with low-latency applications such as web browsing. SkypeMorph [30], FreeWave [22], CensorSpoofer [43] and StegoTorus [44] are proxies or proxy-transports that attempt to mimic other protocols, such as Skype, VoIP, or HTTP in order to avoid censorship by traffic fingerprinting. However, recent work appears to suggest that such mimicry may be detectable under certain circumstances by an adversary [18, 20]. Finally, browser-based proxies work by running a small flash proxy inside non-censored users browsers (for example, when they visit a website), and serve as short-lived proxies for cen-

sored users [17]. These rapidly changing proxies can be difficult for a censor to block in practice, though it is essentially a more fast-paced version of the traditional censor cat-and-mouse game.

Related steganographic techniques Other techniques [3, 6, 31] leverage pseudorandom public-key encryption (i.e., encryption that produces ciphertext indistinguishable from random bits) in order to solve the classic prisoners' problem. These techniques allow protocol participants to produce messages that mimic the distribution of an "innocent-looking" communication channel. The problem setting differs from ours, however, and the encoding of hidden messages inside an allowed encrypted channel (as valid ciphertexts) is not considered.

Dyer et al. [16] introduce a related technique called format transforming encryption (FTE), which disguises encrypted application-layer traffic to look like an innocent, allowed protocol from the perspective of deep packet inspection (DPI) technologies. The basic notion is to transform ciphertexts to match an expected format; as DPI technologies typically use membership in a regular language to classify application-layer traffic, FTE works by using a (bijective) encoding function that maps a ciphertext to a member of a pre-specified language. This steganographic technique differs significantly from ours, in that we do not attempt to disguise the use of a particular internet protocol itself (i.e., TLS), but rather ensure that our encoded ciphertext does not alter the expected distribution of the selected protocol traffic (i.e., TLS ciphertexts, in our system design).

11 Conclusion

End-to-middle proxies are a promising concept that may help tilt the balance of power from censors to citizens. Although previous designs including Telex, Cirripede, and Decoy Routing have laid the ground for this new direction, there are several problems when it comes to deploying any of these designs in practice. Previous designs have required inline blocking elements and sometimes assumed symmetric network paths. To address these concerns, we have developed TapDance, a novel end-to-middle proxy that operates without the need for inline flow blocking. We also described a novel way to support asymmetric flows without inline-flow blocking, by encoding arbitrary-length steganographic payloads in ciphertext. This covert channel may be independently useful for future E2M schemes and other censorship resistance applications.

Ultimately, anticensorship proxies are only useful if they are actually deployed. We hope that removing these barriers to end-to-middle proxying is a step towards that goal.

Acknowledgments

The authors thank Joe Adams, Karl Fogel, Derek Harkness, Steven Kent, Michael Milliken, David Robinson, Steve Schultze, Bob Stovall, Stelios Valavanis, and James Vasile for helpful discussions and encouragement. We also thank Roger Dingledine and the anonymous reviewers. Eric Wustrow conducted this research as an OpenITP Scholar at the New America Foundation. This work was supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1255153 and CNS-1345254 and by an NSF Graduate Research Fellowship.

References

[1] GoAgent open source project. https://code.google.com/p/goagent/.

[2] Ultrasurf. https://ultrasurf.us/.

[3] L. Ahn and N. J. Hopper. Public-key steganography. In *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 323–341. Springer Berlin Heidelberg, 2004.

[4] R. J. Anderson and F. A. P. Petitcolas. On the limits of steganography. *IEEE J. Sel. A. Commun.*, 16(4):474–481, Sept. 2006.

[5] S. Aryan, H. Aryan, and J. A. Halderman. Internet censorship in Iran: A first look. In *3rd USENIX Workshop on Free and Open Communications on the Internet – FOCI '13*. USENIX Association, 2013.

[6] M. Backes and C. Cachin. Public-key steganography with active attacks. In *Theory of Cryptography Conference – TCC '05*, volume 3378 of *LNCS*, pages 210–226. Springer Berlin Heidelberg, 2005.

[7] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer Berlin Heidelberg, 2006.

[8] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *ACM Conference on Computer and Communications Security – CCS 2013*, pages 967–980. ACM, 2013.

[9] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Playstation 3 computing breaks 2^{60} barrier 112-bit prime ECDLP solved. http://lacal.epfl.ch/112bit_prime, 2009.

[10] S. Burnett, N. Feamster, and S. Vempala. Chipping away at censorship firewalls with user-generated content. In *19th USENIX Security Symposium*, pages 463–468. USENIX Association, 2010.

[11] J. Cesareo, J. Karlin, J. Rexford, and M. Schapira. Optimizing the placement of implicit proxies. http://www.cs.princeton.edu/~jrex/papers/decoy-routing.pdf, June 2012.

[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[13] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium*, pages 21–21. USENIX Association, 2004.

[14] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *Internet Measurement Conference – IMC '13*, pages 291–304. ACM, 2013.

[15] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, pages 605–619. USENIX Association, Aug. 2013.

[16] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *ACM Conference on Computer and Communications Security – CCS 2013*, pages 61–72. ACM, 2013.

[17] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies – PETS 2012*, volume 7384 of *LNCS*, pages 239–258. Springer Berlin Heidelberg, 2012.

[18] J. Geddes, M. Schuchard, and N. Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In *ACM Conference on Computer and Communications Security – CCS 2013*, pages 361–372. ACM, 2013.

[19] T. G. Handel and M. T. Sandford, II. Hiding data in the OSI network model. In *Information Hiding – IH '96*, volume 1174 of *LNCS*, pages 23–38. Springer Berlin Heidelberg, 1996.

[20] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *IEEE Symposium on Security and Privacy – SP '13*, pages 65–79. IEEE, 2013.

[21] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *ACM Conference on Computer and Communications Security – CCS 2011*, pages 187–200. ACM, 2011.

[22] A. Houmansadr, T. Riedl, N. Borisov, and A. Singer. I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention. In *Network and Distributed System Security Symposium – NDSS 2013*. Internet Society, 2013.

[23] A. Houmansadr, E. L. Wong, and V. Shmatikov. No direction home: The true cost of routing around decoys. In *Network and Distributed System Security Symposium – NDSS '14*. Internet Society, 2014.

[24] L. Invernizzi, C. Kruegel, and G. Vigna. Message in a bottle: Sailing past censorship. In *29th Annual Computer Security Applications Conference – ACSAC 2013*, pages 39–48. ACM, 2013.

[25] J. Jia and P. Smith. Psiphon: Analysis and estimation. http://www.cdf.toronto.edu/~csc494h/reports/2004-fall/psiphon_ae.html, Oct. 2004.

[26] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer. Decoy routing: Toward unblockable Internet communication. In *USENIX Workshop on Free and Open Communications on the Internet – FOCI '11*. USENIX Association, 2011.

[27] J. Kasten, E. Wustrow, and J. A. Halderman. CAge: Taming certificate authorities by inferring restricted scopes. In *Financial Cryptography and Data Security – FC 2013*, volume 7859 of *LNCS*, pages 329–337. Springer Berlin Heidelberg, 2013.

[28] A. Langley. TLS symmetric crypto. https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html, Feb. 2014.

[29] N. Mathewson and N. Provos. libevent: An event notification library. http://libevent.org/.

[30] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. SkypeMorph: Protocol obfuscation for Tor bridges. In *ACM Conference on Computer and Communications Security – CCS 2012*, pages 97–108. ACM, 2012.

[31] B. Möller. A public-key encryption scheme with pseudo-random ciphertexts. In *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer Berlin Heidelberg, 2004.

[32] S. J. Murdoch and S. Lewis. Embedding covert channels into TCP/IP. In *Information Hiding – IH '05*, volume 3727 of *LNCS*, pages 247–261. Springer Berlin Heidelberg, 2005.

[33] ntop.org. PF_RING: High-speed packet capture, filtering and analysis. http://www.ntop.org/products/pf_ring/.

[34] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[35] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[36] O. Project. OpenSSL: Cryptography and SSL/TLS toolkit. http://www.openssl.org/.

[37] D. Robinson, H. Yu, and A. An. Collateral freedom: A snapshot of Chinese Internet users circumventing censorship. Open Internet Tools Project, Apr. 2013. https://openitp.org/pdfs/CollateralFreedom.pdf.

[38] P. Rogaway. Evaluation of some blockcipher modes of operation. Technical report, Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan, Feb. 2011.

[39] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), Aug. 2008.

[40] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper. Routing around decoys. In *ACM Conference on Computer and Communications Security – CCS 2012*, pages 85–96. ACM, 2012.

[41] G. J. Simmons. The prisoners' problem and the subliminal channel. In *CRYPTO '83*, pages 51–67. Springer US, 1984.

[42] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against SSL (short paper). In *Financial Cryptography and Data Security – FC 2011*, volume 7035 of *LNCS*, pages 250–259. Springer Berlin Heidelberg, 2012.

[43] Q. Wang, X. Gong, G. T. K. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric communication using ip spoofing for censorship-resistant web browsing. In *ACM Conference on Computer and Communications Security – CCS 2012*, pages 121–132. ACM, 2012.

[44] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus: A camouflage proxy for the Tor anonymity system. In *ACM Conference on Computer and Communications Security – CCS 2012*, pages 109–120. ACM, 2012.

[45] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference – ATC '08*, pages 321–334. USENIX Association, 2008.

[46] T. Wilde. Great Firewall Tor probing. https://gist.github.com/twilde/da3c7a9af01d74cd7de7, 2012.

[47] P. Winter and S. Linkdskog. How the Great Firewall of China is blocking Tor. In *2nd USENIX Workshop on Free and Open Communications on the Internet – FOCI '12*, 2012.

[48] J. Wolfgang, M. Dusi, and K. C. Claffy. Estimating routing symmetry on single links by passive flow measurements. pages 473–478. ACM, 2010.

[49] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the network infrastructure. In *20th USENIX Security Symposium*, pages 459–474. USENIX Association, Aug. 2011.

[50] X. Xu, Z. Mao, and J. Halderman. Internet censorship in China: Where does the filtering occur? In *12th Passive and Active Measurement Conference – PAM 2011*, volume 6579 of *LNCS*, pages 133–142, 2011.

A Bayesian Approach to Privacy Enforcement in Smartphones

Omer Tripp
IBM Research, USA

Julia Rubin
IBM Research, Israel

Abstract

Mobile apps often require access to private data, such as the device ID or location. At the same time, popular platforms like Android and iOS have limited support for user privacy. This frequently leads to unauthorized disclosure of private information by mobile apps, e.g. for advertising and analytics purposes. This paper addresses the problem of privacy enforcement in mobile systems, which we formulate as a classification problem: When arriving at a privacy sink (e.g., database update or outgoing web message), the runtime system must classify the sink's behavior as either legitimate or illegitimate. The traditional approach of information-flow (or taint) tracking applies "binary" classification, whereby information release is legitimate iff there is no data flow from a privacy source to sink arguments. While this is a useful heuristic, it also leads to false alarms.

We propose to address privacy enforcement as a learning problem, relaxing binary judgments into a quantitative/probabilistic mode of reasoning. Specifically, we propose a Bayesian notion of statistical classification, which conditions the judgment whether a release point is legitimate on the evidence arising at that point. In our concrete approach, implemented as the BAYESDROID system that is soon to be featured in a commercial product, the evidence refers to the similarity between the data values about to be released and the private data stored on the device. Compared to TaintDroid, a state-of-the-art taint-based tool for privacy enforcement, BAYESDROID is substantially more accurate. Applied to 54 top-popular Google Play apps, BAYESDROID is able to detect 27 privacy violations with only 1 false alarm.

1 Introduction

Mobile apps frequently demand access to private information. This includes unique device and user identifiers, such as the phone number or IMEI number (identifying the physical device); social and contacts data; the user's location; audio (microphone) and video (camera) data; etc. While private information often serves the core functionality of an app, it may also serve other purposes, such as advertising, analytics or cross-application profiling [9]. From the outside, the user is typically unable to distinguish legitimate usage of their private information from illegitimate scenarios, such as sending of the IMEI number to a remote advertising website to create a persistent profile of the user.

Existing platforms provide limited protection against privacy threats. Both the Android and the iOS platforms mediate access to private information via a permission model. Each permission is mapped to a designated resource, and holds per all application behaviors and resource accesses. In Android, permissions are given or denied at installation time. In iOS, permissions are granted or revoked upon first access to the respective resource. Hence, both platforms cannot disambiguate legitimate from illegitimate usage of a resource once an app is granted the corresponding permission [8].

Threat Model In this paper, we address privacy threats due to authentic (as opposed to malicious) mobile applications [4, 18]. Contrary to malware, such applications execute their declared functionality, though they may still expose the user to unnecessary threats by incorporating extraneous behaviors — neither required by their core business logic nor approved by the user [11] — such as analytics, advertising, cross-application profiling, social computing, etc. We consider unauthorized release of private information that (almost) unambiguously identifies the user as a privacy threat. Henceforth, we dub such threats *illegitimate*.

While in general there is no bullet-proof solution for privacy enforcement that can deal with any type of covert channel, implicit flow or application-specific data transformation, and even conservative enforcement approaches can easily be bypassed [19], there is strong evidence that authentic apps rarely exhibit these challenges.

According to a recent study [9], and also our empirical data (presented in Section 5), private information is normally sent to independent third-party servers. Consequently, data items are released in clear form, or at most following well-known encoding/encryption transformations (like Base64 or MD5), to meet the requirement of a standard and general client/server interface.

The challenge, in this setting, is to determine whether the app has taken sufficient means to protect user privacy. Release of private information, even without user authorization, is still legitimate if only a small amount of information has been released. As an example, if an application obtains the full location of the user, but then releases to an analytics server only coarse information like the country or continent, then in most cases this would be perceived as legitimate.

Privacy Enforcement via Taint Analysis The shortcomings of mobile platforms in ensuring user privacy have led to a surge of research on realtime privacy monitoring. The foundational technique grounding this research is *information-flow tracking*, often in the form of *taint analysis* [23, 15]: Private data, obtained via privacy *sources* (e.g. `TelephonyManager.getSubscriberId()`, which reads the device's IMSI), is labeled with a taint tag denoting its source. The tag is then propagated along data-flow paths within the code. Any such path that ends up in a release point, or privacy *sink* (e.g. `WebView.loadUrl(...)`, which sends out an HTTP request), triggers a leakage alarm.

The tainting approach effectively reduces leakage judgments to boolean reachability queries. This can potentially lead to false reports, as the real-world example shown in Figure 1 illustrates. This code fragment, extracted from a core library in the Android platform, reads the device's IMSI number, and then either (ii) persists the full number to an error log if the number is invalid (the `loge(...)` call), or (ii) writes a prefix of the IMSI (of length 6) to the standard log while carefully masking away the suffix (of length 9) as `'x'` characters. Importantly, data flow into the `log(...)` sink is not a privacy problem, because the first 6 digits merely carry model and origin information. Distinctions of this sort are beyond the discriminative power of taint analysis [26].

Quantitative extensions of the core tainting approach have been proposed to address this limitation. A notable example is McCamant and Ernst's [13] information-flow tracking system, which quantities flow of secret information by dynamically tracking taint labels at the bit level. Other approaches — based e.g. on distinguishability between secrets [1], the rate of data transmission [12] or the influence inputs have on output values [14] — have also been proposed. While these systems are useful as offline analyses, it is highly unlikely that any of them can be engineered to meet the performance requirements of a realtime monitoring solution due to the high complexity of their underlying algorithms. As an example, McCamant and Ernst report on a workload on which their analysis spent over an hour.

```
1 String mImsi = ...; // source
2 // 6 digits <= IMSI (MCC+MNC+MSIN) <= 15 (usually 15)
3 if (mImsi != null &&
4     (mImsi.length() < 6 || mImsi.length() > 15)) {
5   loge(" invalid IMSI " + mImsi); // sink
6   mImsi = null; }
7 log("IMSI: " + mImsi.substring(0, 6) + "xxxxxxxxx"); // sink
```

Figure 1: Fragment from an internal Android library, `com.android.internal.telephony.cdma.RuimRecords`, where a prefix of the mobile device's IMSI number flows into the standard log file

Our Approach We formulate data leakage as a classification problem, which generalizes the source/sink reachability judgment enforced by standard information-flow analysis, permitting richer and more relaxed judgments in the form of statistical classification. The motivating observation is that reasoning about information release is fuzzy in nature. While there are clear examples of legitimate versus illegitimate information release, there are also less obvious cases (e.g., a variant of the example in Figure 1 with a 10- rather than 6-character prefix). A statistical approach, accounting for multiple factors and based on rich data sets, is better able to address these subtleties.

Concretely, we propose Bayesian classification. To label a release point as either legitimate or illegitimate, the Bayesian classifier refers to the "evidence" at that point, and computes the likelihood of each label given the evidence. The evidence consists of feature/value pairs. There are many ways of defining the evidence. In this study, we concentrate on the data arguments flowing into release operations, though we intend to consider other classes of features in the future. (See Section 7.)

Specifically, we induce features over the private values stored on the device, and evaluate these features according to the level of similarity between the private values and those arising at release points. This distinguishes instances where data that is dependent on private values flows into a release point, but its structural and/or quantitative characteristics make it eligible for release, from illegitimate behaviors. Failure to make such distinctions is a common source of false alarms suffered by the tainting approach [4].

To illustrate this notion of features, we return to the example in Figure 1. Because the IMSI number is consid-

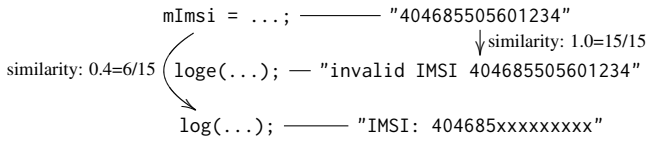


Figure 2: Similarity analysis applied to the code in Figure 1

ered private, we define a respective feature *IMSI*. Assume that the concrete IMSI value is "404685505601234". Then the value arising at the `log(...)` release point is "IMSI: 404685xxxxxxxxx". The quantitative similarity between these two values serves as evidence for the decision whether or not `log(...)` is behaving legitimately. This style of reasoning is depicted in Figure 2.

Evaluation To evaluate our approach, we have implemented the BAYESDROID system for privacy enforcement. We report on two sets of experiments over BAYESDROID.

First, to measure the accuracy gain thanks to Bayesian analysis, we compared BAYESDROID with the TaintDroid system [4], a highly popular and mature implementation of the tainting approach that is considered both efficient (with average overhead of approximately 10%) and accurate. We applied both BAYESDROID and TaintDroid to the DroidBench suite,[1] which comprises the most mature and comprehensive set of privacy benchmarks currently available. The results suggest dramatic improvement in accuracy thanks to Bayesian elimination of false reports, yielding accuracy scores of 0.96 for BAYESDROID versus 0.66 for TaintDroid.

The second experiment examines the practical value of BAYESDROID by applying it to 54 top-popular mobile apps from Google Play. We evaluate two variants of BAYESDROID, one of which is able to detect a total of 27 distinct instances of illegitimate information release across 15 of the applications with only 1 false alarm.

Contributions This paper makes the following principal contributions:

1. Novel approach to leakage detection (Section 2): We present a Bayesian classification alternative to the classic tainting approach. Our approach is more flexible than taint tracking by permitting statistical weighting of different features as the basis for privacy judgments.
2. Similarity-based reasoning (Section 3): We instantiate the Bayesian approach by applying quantitative similarity judgments over private values and values about to be released. This enables consideration of actual data, rather than only data flow, as evidence for privacy judgments.
3. Implementation and evaluation (Sections 4–5): We have instantiated our approach as the BAYESDROID system, which is about to be featured in an IBM cloud-based security service. We report on two sets of experiments, whose results (i) demonstrate substantial accuracy gain thanks to Bayesian reasoning, and (ii) substantiate the overall effectiveness of BAYESDROID when applied to real-world apps. All the leakage reports by BAYESDROID are publicly available for scrutiny.[2]

2 The Bayesian Setting

Our starting point is to treat privacy enforcement as a classification problem, being the decision whether or not a given release point is legitimate. The events, or instances, to be classified are (runtime) release points. The labels are *legitimate* and *illegitimate*. Misclassification either yields a false alarm (mistaking benign information release as a privacy threat) or a missed data leak (failing to intercept illegitimate information release).

2.1 Bayes and Naive Bayes

Our general approach is to base the classification on the *evidence* arising at the release point. Items of evidence may refer to qualitative facts, such as source/sink dataflow reachability, as well as quantitative measures, such as the degree of similarity between private values and values about to be released. These latter criteria are essential in going beyond the question of *whether* private information is released to also reason about the *amount* and *form* of private information about to be released.

A popular classification method, representing this mode of reasoning, is based on Bayes' theorem (or rule). Given events X and Y, Bayes' theorem states the following equality:

$$\Pr(Y|X) = \frac{\Pr(X|Y)\cdot\Pr(Y)}{\Pr(X)} \qquad (1)$$

where $\Pr(Y|X)$ is the conditional probability of Y given X (i.e., the probability for Y to occur given that X has occurred). X is referred to as the *evidence*. Given evidence X, Bayesian classifiers compute the conditional likelihood of each label (in our case, *legitimate* and *illegitimate*).

We begin with the formal background by stating Equation 1 more rigorously. Assume that Y is a discrete-valued random variable, and let $X = [X_1, \ldots, X_n]$ be a

[1] http://sseblog.ec-spride.de/tools/droidbench/

[2] researcher.ibm.com/researcher/files/us-otripp/Artifacts.zip

vector of n discrete or real-valued attributes X_i. Then

$$\Pr(Y = y_k | X_1 \ldots X_n) = \frac{\Pr(Y = y_k) \cdot \Pr(X_1 \ldots X_n | Y = y_k)}{\Sigma_j \Pr(Y = y_j) \cdot \Pr(X_1 \ldots X_n | Y = y_j)} \quad (2)$$

As Equation 2 hints, training a Bayesian classifier is, in general, impractical. Even in the simple case where the evidence X is a vector of n boolean attributes and Y is boolean, we are still required to estimate a set

$$\theta_{ij} = \Pr(X = x_i | Y = y_j)$$

of parameters, where i assumes 2^n values and j assumes 2 values for a total of $2 \cdot (2^n - 1)$ independent parameters.

Naive Bayes deals with the intractable sample complexity by introducing the assumption of conditional independence, as stated in Definition 2.1 below, which reduces the number of independent parameters sharply to $2n$. Intuitively, conditional independence prescribes that events X and Y are independent given knowledge that event Z has occurred.

Definition 2.1 (Conditional Independence)**.** *Given random variables X, Y and Z, we say that X is* conditionally independent *of Y given Z iff the probability distribution governing X is independent of the value of Y given Z. That is,*

$$\forall i, j, k.\ \Pr(X = x_i | Y = y_j, Z = z_k) = \Pr(X = x_i | Z = z_k)$$

Under the assumption of conditional independence, we obtain the following equality:

$$\Pr(X_1 \ldots X_n | Y) = \Pi_{i=1}^{n} \Pr(X_i | Y) \quad (3)$$

Therefore,

$$\Pr(Y = y_k | X_1 \ldots X_n) = \frac{\Pr(Y = y_k) \cdot \Pi_i \Pr(X_i | Y = y_k)}{\Sigma_j \Pr(Y = y_j) \cdot \Pi_i \Pr(X_i | Y = y_j)} \quad (4)$$

2.2 Bayesian Reasoning about Leakage

For leakage detection, conditional independence translates into the requirement that at a release point *st*, the "weight" of evidence e_1 is not affected by the "weight" of evidence e_2 knowing that *st* is legitimate/illegitimate. As an example, assuming the evidence is computed as the similarity between private and released values, if *st* is known to be a statement sending private data to the network, then the similarity between the IMSI number and respective values about to be released is assumed to be independent of the similarity between location coordinates and respective values about to be released.

The assumption of conditional independence induces a "modular" mode of reasoning, whereby the privacy features comprising the evidence are evaluated independently. This simplifies the problem of classifying a release point according to the Bayesian method into two quantities that we need to clarify and estimate: (i) the likelihood of legitimate/illegitimate release ($\Pr(Y = y_k)$) and (ii) the conditional probabilities $\Pr(X_i | Y = y_k)$.

3 Privacy Features

In this section we develop, based on the mathematical background in Section 2, an algorithm to compute the conditional likelihood of legitimate versus illegitimate data release given privacy features F_i. With such an algorithm in place, given values v_i for the features F_i, we obtain

$$v_{leg} = \Pr(legitimate | [F_1 = v_1, \ldots, F_n = v_n])$$
$$v_{illeg} = \Pr(illegitimate | [F_1 = v_1, \ldots, F_n = v_n])$$

Bayesian classification then reduces to comparing between v_{leg} and v_{illeg}, where the label corresponding to the greater of these values is the classification result.

3.1 Feature Extraction

The first challenge that arises is how to define the features (denoted with italicized font: F) corresponding to the private values (denoted with regular font: F). This requires simultaneous consideration of both the actual private value and the "relevant" values arising at the sink statement (or release point). We apply the following computation:

1. Reference value: We refer to the actual private value as the *reference value*, denoting the value of private item F as $[\![\mathrm{F}]\!]$. For the example in Figures 1–2, the reference value, $[\![\mathrm{IMSI}]\!]$, of the *IMSI* feature would be the device's IMSI number: $[\![\mathrm{IMSI}]\!]$ = "404685505601234".
2. Relevant value: We refer to value v about to be released by the sink statement as *relevant* with respect to feature F if there is data-flow connectivity between a source statement reading the value $[\![\mathrm{F}]\!]$ of F and v. Relevant values can thus be computed via information-flow tracking by propagating a unique tag (or label) per each private value, as tools like TaintDroid already do. Note that for a given feature F, multiple different relevant values may arise at a given sink statement (if the private item F flows into more than one sink argument).
3. Feature value: Finally, given the reference value $[\![\mathrm{F}]\!]$ and a set $\{v_1, \ldots, v_k\}$ of relevant values for feature F, the value we assign to F (roughly) reflects the highest degree of pairwise similarity (i.e., minimal

distance) between $[\![\mathrm{F}]\!]$ and the values v_i. Formally, we assume a distance metric d. Given d, we define:

$$[\![F]\!] \equiv \min_{1 \le i \le k} \{d([\![\mathrm{F}]\!], v_i)\}$$

We leave the distance metric $d(\ldots)$ unspecified for now, and return to its instantiation in Section 3.2.

According to our description above, feature values are unbounded in principle, as they represent the distance between the reference value and any data-dependent sink values. In practice, however, assuming (i) the distance metric $d(\ldots)$ satisfies $d(x,y) \le \max\{|x|,|y|\}$, (ii) $\exists c \in \mathbb{N}.\ |[\![\mathrm{F}]\!]| \le c$ (as with the IMEI, IMSI, location, etc.), and (iii) $[\![\mathrm{F}]\!]$ is not compared with values larger than it, we can bound $[\![F]\!]$ by c. In general, any feature can be made finite, with (at most) $n+1$ possible values, by introducing a privileged "$\ge n$" value, which denotes that the distance between the reference and relevant values is at least n.

3.2 Measuring Distance between Values

To compute a quantitative measure of similarity between data values, we exploit the fact that private data often manifests as strings of ASCII characters [4, 9, 27]. These include e.g. device identifiers (like the IMEI and IMSI numbers), GPS coordinates, inter-application communication (IPC) parameters, etc. This lets us quantify distance between values in terms of string metrics.

Many string metrics have been proposed to date [17]. Two simple and popular metrics, which we have experimented with and satisfy the requirement that $d(x,y) \le \max\{|x|,|y|\}$, are the following:

Hamming Distance This metric assumes that the strings are of equal length. The Hamming distance between two strings is equal to the number of positions at which the corresponding symbols are different (as indicated by the indicator function $\delta_{c_1 \ne c_2}(\ldots)$):

$$\mathsf{ham}(a,b) = \Sigma_{0 \le i < |a|} \delta_{c_1 \ne c_2}(a(i), b(i))$$

In another view, Hamming distance measures the number of substitutions required to change one string into the other.

Levenshtein Distance The Levenshtein string metric computes the distance between strings a and b as $\mathsf{lev}_{a,b}(|a|,|b|)$ (abbreviated as $\mathsf{lev}(|a|,|b|)$), where

$$\mathsf{lev}(i,j) = \begin{cases} \max(i,j) & \textit{if } \min(i,j) = 0 \\ \min \begin{pmatrix} \mathsf{lev}(i-1,j)+1 \\ \mathsf{lev}(i,j-1)+1 \\ \mathsf{lev}(i-1,j-1)+\delta_{a_i \ne b_j} \end{pmatrix} & \textit{otherwise} \end{cases}$$

Informally, $\mathsf{lev}(|a|,|b|)$ is the minimum number of single-character edits — either insertion or deletion or substitution — needed to transform one string into the other. An efficient algorithm for computing the Levenshtein distance is bottom-up dynamic programming [24]. The asymptotic complexity is $O(|a| \cdot |b|)$.

Data: Strings u and v
Data: Distance metric d

```
begin
    x ⟵ |u| < |v| ? u : v // min
    y ⟵ |u| ≥ |v| ? u : v // max
    r ⟵ y
    for i = 0 to |y| − |x| do
        y′ ⟵ y[i, i + |x| − 1]
        if d(x, y′) < r then
            r ⟵ d(x, y′)
        end
    end
    return r
end
```

Algorithm 1: The BAYESDROID distance measurement algorithm

Given string metric $d(x,y)$ and pair (u,v) of reference value u and relevant value v, BAYESDROID computes their distance according to the following steps:

1. BAYESDROID ensures that both u and v are `String` objects by either (i) invoking `toString()` on reference types or (ii) converting primitive types into `Strings` (via `String.valueOf(...)`), if the argument is not already of type `String`.
2. To meet the conservative requirement that $|x| = |y|$ (i.e., x and y are of equal length), BAYESDROID applies Algorithm 1. This algorithm induces a sliding window over the longer of the two strings, whose width is equal to the length of the shorter string. The shorter string is then compared to contiguous segments of the longer string that have the same length. The output is the minimum across all comparisons.

To ensure that comparisons are still meaningful under length adjustment, we decompose private values into indivisible *information units*. These are components of the private value that cannot be broken further, and so comparing them with a shorter value mandates that the shorter value be padded. In our specification, the phone, IMEI and IMSI numbers consist of only one unit of information. The `Location` object is an example of a data structure that consists of several distinct information units. These include the integral and fractional parts of the longitude and latitude values, etc. BAYESDROID handles objects that decompose into multiple information units by treating each unit as a separate object and applying the steps above to each unit in turn. The notion of information units guards BAYESDROID against ill-founded judgments, such as treating release of a single IMEI digit as strong evidence for leakage.

3.3 Estimating Probabilities

The remaining challenge, having clarified what the features are and how their values are computed, is to estimate the probabilities appearing in Equation 4:

- We need to estimate the probability of the *legitimate* event, $\Pr(legitimate)$, where *illegitimate* is the complementary event and thus $\Pr(illegitimate) = 1 - \Pr(legitimate)$.
- We need to estimate the conditional probabilities $\Pr(F = u|legitimate)$ and $\Pr(F = u|illegitimate)$ for all features F and respective values u.

$\Pr(legitimate)$ can be approximated straightforwardly based on available statistics on the frequency of data leaks in the wild. For the conditional probabilities, assuming feature X_i is discrete valued with j distinct values (per the discussion in Section 3.1 above), we would naively compute the estimated conditional probability θ_{ijk} according to the following equation:

$$\theta_{ijk} = \widehat{\Pr}(X_i = x_{ij}|Y = y_k) \quad = \quad \frac{\#D\{X_i=x_{ij} \wedge Y=y_k\}}{\#D\{Y=y_k\}} \qquad (5)$$

The danger, however, is that this equation would produce estimates of zero if the data happens not to contain any training examples satisfying the condition in the numerator. To fix this, we modify Equation 5 as follows:

$$\theta_{ijk} = \widehat{\Pr}(X_i = x_{ij}|Y = y_k) \quad = \quad \frac{\#D\{X_i=x_{ij} \wedge Y=y_k\}+l}{\#D\{Y=y_k\}+l \cdot J} \qquad (6)$$

where l is a factor that "smoothens" the estimate by adding in a number of "hallucinated" examples that are assumed to be spread evenly across the J possible values of X_i. In Section 5.1, we provide concrete detail on the data sets and parameter values we used for our estimates.

4 The BAYESDROID Algorithm

In this section, we describe the complete BAYESDROID algorithm. We then discuss enhancements of the core algorithm.

4.1 Pseudocode Description

Algorithm 2 summarizes the main steps of BAYESDROID. For simplicity, the description in Algorithm 2 assumes that source statements serve private data as their return value, though the BAYESDROID implementation also supports other sources (e.g. callbacks like `onLocationChanged(...)`, where the private `Location` object is passed as a parameter). We also assume that each source maps to a unique privacy feature. Hence, when a source is invoked (i.e., the `OnSourceStatement` event fires), we obtain the unique tag corresponding to its respective feature via the `GetFeature(...)` function. We

```
Input: S // privacy specification

begin
    while true do
        OnSourceStatement r := src p̄ :
            // map source to feature
            f ⟵ GetFeature src
            attach tag f to r
        OnNormalStatement r := nrm p̄ :
            propagate feature tags according to data flow
        OnSinkStatement r := snk p̄ :
            // map feat.s to param.s with resp. tag
            {f ↦ p̄_f} ⟵ ExtractTags p̄
            foreach f → p̄_f ∈ {f → p̄_f} do
                u ⟵ ref f
                δ ⟵ min{d(u, [[p]])}_{p∈p̄_f}
                f ⟵ δ ≥ c_f ? "≥_{c_f}" : δ
            end
            if IsLeakageClassification {f} then
                Alarm snk p̄
            end
    end
end
```

Algorithm 2: Outline of the core BAYESDROID algorithm

then attach the tag to the return value r. Normal data flow obeys the standard rules of tag propagation, which are provided e.g. by Enck et al. [4]. (See Table 1 there.)

When an `OnSinkStatement` event is triggered, the arguments flowing into the sink `snk` are searched for privacy tags, and a mapping from features f to parameters p_f carrying the respective tag is built. The value of f is then computed as the minimal pairwise distance between the parameters $p \in p_f$ and `ref` f. If this value is greater than some constant c_f defined for f, then the privileged value "$\geq c_f$" is assigned to f. (See Section 3.1.) Finally, the judgment `IsLeakageClassification` is applied over the features whose tags have reached the sink `snk`. This judgment is executed according to Equation 4.

We illustrate the BAYESDROID algorithm with reference to Figure 3, which demonstrates a real leakage instance in `com.g6677.android.princesshs`, a popular gaming application. In this example, two different private items flow into the sink statement: both the IMEI, read via `getDeviceId()`, and the Android ID, read via `getString(...)`.

At sink statement `URL.openConnection(...)`, the respective tags *IMEI* and *AndroidID* are extracted. Values are assigned to these features according to the description in Section 3, where we utilize training data, as discussed later in Section 5.1, for Equation 6:

$$\Pr(IMEI \geq 5|leg) = 0.071 \quad \Pr(AndID \geq 5|leg) = 0.047$$
$$\Pr(IMEI \geq 5|ilg) = 0.809 \quad \Pr(AndID \geq 5|ilg) = 0.833$$

```
source : private value
    TelephonyManager.getDeviceId() : 000000000000000
    Settings$Secure.getString (...)  : cdf15124ea4c7ad5

sink : arguments
 URL.openConnection(...) : app_id=2aec0559c930 ... &
 android_id =cdf15124ea4c7ad5 \& udid= ... &
  serial_id = ... & ... &
  publisher_user_id =000000000000000
```

Figure 3: True leakage detected by BAYESDROID in `com.g6677.android.princesshs`

We then compute Equation 4, where the denominator is the same for both *leg* and *illeg*, and so it suffices to evaluate the nominator (denoted with $\tilde{\Pr}(...)$ rather than $\Pr(...)$):

$$\tilde{\Pr}(leg|IMEI \geq 5, AndID \geq 5) =$$
$$\Pr(leg) \times \Pr(IMEI \geq 5|leg) \times \Pr(AndID \geq 5|leg) =$$
$$0.66 \times 0.071 \times 0.047 = 0.002$$
$$\tilde{\Pr}(ilg|IMEI \geq 5, AndID \geq 5) =$$
$$\Pr(ilg) \times \Pr(IMEI \geq 5|ilg) \times \Pr(AndID \geq 5|ilg) =$$
$$0.33 \times 0.809 \times 0.833 = 0.222$$

Our estimates of 0.66 for $\Pr(leg)$ and 0.33 for $\Pr(ilg)$ are again based on training data as explained in Section 5.1. The obtained conditional measure of 0.222 for *ilg* is (far) greater than 0.002 for *leg*, and so BAYESDROID resolves the release instance in Figure 3 as a privacy threat, which is indeed the correct judgment.

4.2 Enhancements

We conclude our description of BAYESDROID by highlighting two extensions of the core algorithm.

Beyond Plain Text While many instances of illegitimate information release involve plain text, and can be handled by the machinery in Section 3.1, there are also more challenging scenarios. Two notable challenges are (i) data transformations, whereby data is released following an encoding, encryption or hashing transformation; and (ii) high-volume binary data, such as camera or microphone output. We have extended BAYESDROID to address both of these cases.

We begin with data transformations. As noted earlier, in Section 1, private information is sometimes released following standard hashing/encoding transformations, such as the Base64 scheme. This situation, illustrated in Figure 4, can distort feature values, thereby

```
TelephonyManager tm =
    getSystemService(TELEPHONY_SERVICE);
String imei = tm.getDeviceId(); // source
String encodedIMEI = Base64Encoder.encode(imei);
Log.i(encodedIMEI); // sink
```

Figure 4: Adaptation of the DroidBench `Loop1` benchmark, which releases the device ID following Base64 encoding

leading BAYESDROID to erroneous judgments. Fortunately, the transformations that commonly manifest in leakage scenarios are all standard, and there is a small number of such transformations [9].

To account for these transformations, BAYESDROID applies each of them to the value obtained at a source statement, thereby exploding the private value into multiple representations. This is done lazily, once a sink is reached, for performance. This enhancement is specified in pseudocode form in Algorithm 3. The main change is the introduction of a loop that traverses the transformations $\tau \in T$, where the identity transformation, $\lambda x.\ x$, is included to preserve the (non-transformed) value read at the source. The value assigned to feature f is then the minimum with respect to all transformed values.

Binary data — originating from the microphone, camera or bluetooth adapter — also requires special handling because of the binary versus ASCII representation and, more significantly, its high volume. Our solution is guided by the assumption that such data is largely treated as "uninterpreted" and immutable by application code due to its form and format. This leads to a simple yet effective strategy for similarity measurement, whereby a fixed-length prefix is truncated out of the binary content. Truncation is also applied to sink arguments consisting of binary data.

Heuristic Detection of Relevant Values So far, our description of the BAYESDROID algorithm has relied on tag propagation to identify relevant values at the sink statement. While this is a robust mechanism to drive feature computation, flowing tags throughout the code also has its costs, incurring runtime overheads of $\geq 10\%$ and affecting the stability of the application due to intrusive instrumentation [4].

These weaknesses of the tainting approach have led us to investigate an alternative method of detecting relevant values. A straightforward relaxation of data-flow tracking is bounded ("brute-force") traversal of the reachable values from the arguments to a sink statement up to some depth bound k: All values pointed-to by a sink argument or reachable from a sink argument via a sequence of $\leq k$ field dereferences are deemed relevant. Though in theory

Input: $T \equiv \{\lambda x.\, x, \tau_1, \ldots, \tau_n\}$ `// std. transformations`

begin
  ...
  `OnSinkStatement r := snk` $\overline{p}$:
    $\{f \mapsto \overline{p_f}\} \longleftarrow$ `ExtractTags` $\overline{p}$
    foreach $f \to \overline{p_f} \in \{f \to \overline{p_f}\}$ **do**
      foreach $\tau \in T$ **do**
        $u \longleftarrow \tau$ (`ref` f)
        $\delta \longleftarrow \min\{d(u, [\![p]\!])\}_{p \in \overline{p_f}}$
        $f \longleftarrow \min\{[\![f]\!], \delta \geq c_f \; ? \text{“} \geq_{c_f} \text{”} : \delta\}$
      end
    end
  ...
end

Algorithm 3: BAYESDROID support for standard data transformations

this might introduce both false positives (due to irrelevant values that are incidentally similar to the reference value) and false negatives (if k is too small, blocking relevant values from view), in practice both are unlikely, as we confirmed experimentally. (See Section 5.)

For false positives, private values are often unique, and so incidental similarity to irrelevant values is improbable. For false negatives, the arguments flowing into privacy sinks are typically either `String` objects or simple data structures. Also, because the number of privacy sinks is relatively small, and the number of complex data structures accepted by such sinks is even smaller, it is possible to specify relevant values manually for such data structures. We have encountered only a handful of data structures (e.g. the `android.content.Intent` class) that motivate a specification.

5 Experimental Evaluation

In this section, we describe the BAYESDROID implementation, and present two sets of experiments that we have conducted to evaluate our approach.

5.1 The BAYESDROID System

Implementation Similarly to existing tools like TaintDroid, BAYESDROID is implemented as an instrumented version of the Android SDK. Specifically, we have instrumented version 4.1.1_r6 of the SDK, which was chosen intentionally to match the most recent version of TaintDroid.[3] The experimental data we present indeed utilizes TaintDroid for tag propagation (as required for accurate resolution for relevant values).

[3] http://appanalysis.org/download.html

Beyond the TaintDroid instrumentation scheme, the BAYESDROID scheme specifies additional behaviors for sources and sinks within the SDK. At source points, a hook is added to record the private value read by the source statement (which acts as a reference value). At sink points, a hook is installed to apply Bayesian reasoning regarding the legitimacy of the sink.

Analogously to TaintDroid, BAYESDROID performs privacy monitoring over APIs for file-system access and manipulation, inter-application and socket communication, reading the phone's state and location, and sending of text messages. BAYESDROID also monitors the HTTP interface, camera, microphone, bluetooth and contacts. As explained in Section 4.1, each of the privacy sources monitored by BAYESDROID is mirrored by a tag/feature. The full list of features is as follows: *IMEI*, *IMSI*, *AndroidID*, *Location*, *Microphone*, *Bluetooth*, *Camera*, *Contacts* and *FileSystem*.

The BAYESDROID implementation is configurable, enabling the user to switch between distance metrics as well as enable/disable information-flow tracking for precise/heuristic determination of relevant values. (See Section 4.2.) In our experiments, we tried both the Levenshtein and the Hamming metrics, but found no observable differences, and so we report the results only once. Our reasoning for why the metrics are indistinguishable is because we apply both to equal-length strings (see Section 3.2), and have made sure to apply the same metric both offline and online, and so both metrics achieve a very similar effect in the Bayesian setting.

Training To instantiate BAYESDROID with the required estimates, as explained in Section 3.3, we applied the following methodology: First, to estimate $\Pr(\mathit{legitimate})$, we relied on (i) an extensive study by Hornyack et al. spanning 1,100 top-popular free Android apps [9], as well as (ii) a similarly comprehensive study by Enck et al. [5], which also spans a set of 1,100 free apps. According to the data presented in these studies, approximately one out of three release points is illegitimate, and thus $\widehat{\Pr}(\mathit{legitimate}) = 0.66$ and complementarily $\widehat{\Pr}(\mathit{illegitimate}) = 1 - 0.66 \approx 0.33$.

For the conditional probabilities $\widehat{\Pr}(X_i = x_{ij} | Y = y_k)$, we queried Google Play for the 100 most popular apps (across all domains) in the geography of one of the authors. We then selected at random 35 of these apps, and analyzed their information-release behavior using debug breakpoints (which we inserted via the `adb` tool that is distributed as part of the Android SDK).

Illegitimate leaks that we detected offline mainly involved (i) location information and (ii) device and user identifiers, which is consistent with the findings reported by past studies [9, 5]. We confirmed that illegitimate leaks are largely correlated with high similarity between

private data and sink arguments, and so we fixed six distance levels for each private item: $[0,4]$ and "≥ 5". (See Section 3.1.) Finally, to avoid zero estimates for conditional probabilities while also minimizing data perturbation, we set the "smoothing" factor l in Equation 6 at 1, where the illegitimate flows we detected were in the order of several dozens per private item.

5.2 Experimental Hypotheses

In our experimental evaluation of BAYESDROID, we tested two hypotheses:

1. H1: Accuracy. Bayesian reasoning, as implemented in BAYESDROID, yields a significant improvement in leakage-detection accuracy compared to the baseline of information-flow tracking.
2. H2: Applicability. For real-life applications, BAYESDROID remains effective under relaxation of the tag-based method for detection of relevant values and its stability improves.

5.3 H1: Accuracy

To assess the accuracy of BAYESDROID, we compared it to that of TaintDroid, a state-of-the-art information-flow tracking tool for privacy enforcement. Our experimental settings and results are described below.

Subjects We applied both TaintDroid and BAYESDROID to DroidBench, an independent and publicly available collection of benchmarks serving as testing ground for both static and dynamic privacy enforcement algorithms. DroidBench models a large set of realistic challenges in leakage detection, including precise tracking of sensitive data through containers, handling of callbacks, field and object sensitivity, lifecycle modeling, inter-app communication, reflection and implicit flows.

The DroidBench suite consists of 50 cases. We excluded from our experiment (i) 8 benchmarks that crash at startup, as well as (ii) 5 benchmarks that leak data via callbacks that we did not manage to trigger (e.g., `onLowMemory()`), as both TaintDroid and BAYESDROID were naturally unable to detect leakages in these two cases. The complete list of benchmarks that we used can be found in Table 4 of Appendix B.

Methodology For each benchmark, we measured the number of true positive (TP), false positive (FP) and false negative (FN) results. We then summarized the results and calculated the overall *precision* and *recall* of each tool using the formulas below:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

	TPs	FPs	FNs	Precision	Recall	F-measure
TaintDroid	31	17	0	0.64	1.00	0.78
BAYESDROID	29	1	2	0.96	0.93	0.94

Table 1: Accuracy of BAYESDROID and TaintDroid on DroidBench

High precision implies that a technique returns few irrelevant results, whereas high recall implies that it misses only few relevant ones.

Since ideal techniques have both high recall and high precision, the F-measure is commonly used to combine both precision and recall into a single measure. The F-measure is defined as the harmonic mean of precision and recall, and is calculated as follows:

$$F\text{-}Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The value of F-measure is high only when both precision and recall are high. We thus use the F-measure for accuracy evaluation.

Results The results obtained for both TaintDroid and BAYESDROID on version 1.1 of DroidBench are summarized in Table 1 and presented in detail in Table 4. The findings reported by BAYESDROID are also publicly available.[4]

Overall, TaintDroid detects 31 true leakages while also reporting 17 false positives, whereas BAYESDROID suffers from 2 false negatives, discovering 29 of the true leakages while flagging only 1 false alarm. The recall of both TaintDroid and BAYESDROID is high (1 and 0.93, respectively) due to a low number of false-negative results. Yet the precision of TaintDroid is much lower than that of BAYESDROID (0.64 vs. 0.96), due to a high number of false positives. The overall F-measure is thus lower for TaintDroid than for BAYESDROID (0.78 vs. 0.94).

The results mark BAYESDROID as visibly more accurate than TaintDroid. To further confirm this result, we performed a two-tail McNemar test, considering 48 observations for each tool. These observations correspond to findings reported in Table 4: 31 true positives and 17 classified as false alarms. Each observation is a boolean value that represents the accuracy of the tool and is assumed to be from a Bernoulli distribution. We then checked whether the difference in accuracy is statistically significant by testing the null hypothesis that the set of 48 observations from TaintDroid are sampled from the same Bernoulli distribution as the set of 48 observations from BAYESDROID.

[4] See archive file researcher.ibm.com/researcher/files/us-otripp/droidbench.zip.

```
1 TelephonyManager tm =
2     getSystemService(TELEPHONY_SERVICE);
3 String imei = tm.getDeviceId(); //source
4 String obfuscatedIMEI = obfuscateIMEI(imei); ...;
5 Log.i(imei); // sink
6
7 private String obfuscateIMEI(String imei) {
8   String result = "";
9   for (char c : imei.toCharArray()) {
10    switch(c) {
11      case '0': result += 'a'; break;
12      case '1': result += 'b'; break;
13      case '2': result += 'c'; break; ...; } }
```

Figure 5: Fragment from the DroidBench `ImplicitFlow1` benchmark, which applies a custom transformation to private data

We found that TaintDroid was accurate in 31 out of 48 cases, and BAYESDROID was accurate in 45 out of 48 cases. We built the 2×2 contingency table showing when each tool was correct and applied a two-tail McNemar test. We found a p-value of 0.001, which rejects the null hypothesis that the observations come from the same underlying distribution and provides evidence that BAYESDROID is more accurate than TaintDroid, thereby confirming H1.

Discussion Analysis of the per-benchmark findings reveals the following: First, the 2 false negatives of BAYESDROID on `ImplicitFlow1` are both due to custom (i.e., non-standard) data transformations, which are outside the current scope of BAYESDROID. An illustrative fragment from the `ImplicitFlow1` code is shown in Figure 5. The `obfuscateIMEI(...)` transformation maps IMEI digits to English letters, which is a non-standard behavior that is unlikely to arise in an authentic app.

The false positive reported by BAYESDROID, in common with TaintDroid, is on release of sensitive data to the file system, albeit using the `MODE_PRIVATE` flag, which does not constitute a leakage problem in itself. This can be resolved by performing Bayesian reasoning not only over argument values, but also over properties of the sink API (in this case, the storage location mapped to a file handle). We intend to implement this enhancement.

Beyond the false alarm in common with BAYESDROID, TaintDroid has multiple other sources of imprecision. The main reasons for its false positives are

- coarse modeling of containers, mapping their entire contents to a single taint bit, which accounts e.g. for the false alarms on `ArrayAccess{1,2}` and `HashMapAccess1`;
- field and object insensitivity, resulting in false alarms on `FieldSensitivity{2,4}` and `ObjectSensitivity{1,2}`; and more fundamentally,
- ignoring of data values, which causes TaintDroid to issue false warnings on `LocationLeak{1,2}` even when location reads fail, yielding a `Location` object without any meaningful information.

The fundamental reason for these imprecisions is to constrain the overhead of TaintDroid, such that it can meet the performance demands of online privacy enforcement. BAYESDROID is able to accommodate such optimizations while still ensuring high accuracy.

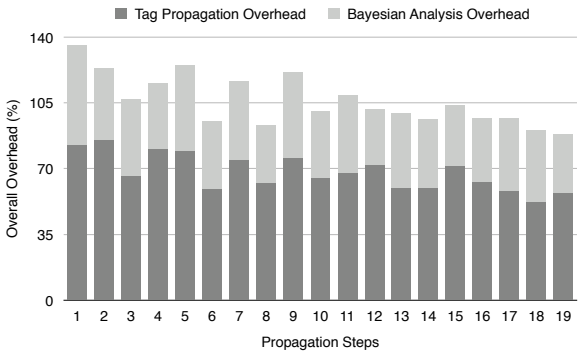


Figure 6: Overhead breakdown into tag propagation and Bayesian analysis at sink

5.4 H2: Applicability

The second aspect of the evaluation compared between two versions of BAYESDROID, whose sole difference lies in the method used for detecting relevant values: In one configuration (T-BD), relevant values are detected via tag propagation. The other configuration (H-BD) uses the heuristic detailed in Section 4.2 of treating all values reachable from sink arguments (either directly or via the heap graph) up to a depth bound of k as relevant, which places more responsibility on Bayesian reasoning. We set k at 3 based on manual review of the data structures flowing into privacy sinks.

We designed a parametric benchmark application to quantify the overhead reduction imposed by the H-BD variant of BAYESDROID. The application consists of a simple loop that flows the device IMEI into a log file. Loop iterations perform intermediate data propagation steps. We then performed a series of experiments — over the range of 1 to 19 propagation steps — to quantify the relative overhead of tag propagation versus Bayesian analysis.

The results, presented in Figure 6, suggest that the overhead of tag propagation is more dominant than that of Bayesian analysis (with a ratio of roughly 2:1), even when the set of relevant values is naively over approximated. Discussion of the methodology underlying this

experiment is provided in Appendix A.

In general, H-BD trades overhead reduction for accuracy. H2 then asserts that, *in practice*, the tradeoff posed by H-BD is effective. Below, we discuss our empirical evaluation of this hypothesis over real-life subjects.

Subjects To avoid evaluators' bias, we applied the following selection process: We started from the 65 Google Play apps not chosen for the training phase. We then excluded 8 apps that do not have permission to access sensitive data and/or perform release operations (i.e., their manifest does not declare sufficient permissions out of `INTERNET`, `READ_PHONE_STATE`, `SEND_SMS`, etc), as well as 3 apps that we did not manage to install properly, resulting in 54 apps that installed successfully and exercise privacy sources and sinks.

The complete list of the application we used is given in Table 5 of Appendix B. A subset of the applications, for which at least one leakage was detected, is also listed in Table 3.

Methodology We deployed the apps under the two BAYESDROID configurations. Each execution was done from a clean starting state. The third column of both Tables 3 and 5 denotes whether our exploration of the app was exhaustive. By that we mean exercising all the UI points exposed by the app in a sensible order. Ideally we would do so for all apps. However, (i) some of the apps, and in particular gaming apps, had stability issues, and (ii) certain apps require SMS-validated sign in, which we did not perform. We did, however, create Facebook, Gmail and Dropbox accounts to log into apps that demand such information yet do not ask for SMS validation. We were also careful to execute the exact same crawling scenario under both the T-BD and H-BD configurations. We comment, from our experience, that most data leaks happen when an app launches, and initializes advertising/analytics functionality, and so for apps for which deep crawling was not possible the results are still largely meaningful.

For comparability between the H-BD and T-BD configurations, we counted different dynamic reports involving the same pair of source/sink APIs as a single leakage instance. We manually classified the findings into true positives and false positives. For this classification, we scrutinized the reports by the two configurations, and also — in cases of uncertainty — decompiled and/or reran the app to examine its behavior more closely. As in the experiment described in Section 5.3, we then calculated the precision, recall and F-measure for each of the tools.

	TPs	FPs	FNs	Precision	Recall	F-measure	Crashes
H-BD	27	1	0	0.96	1.00	0.98	12
T-BD	14	0	10	1.00	0.58	0.73	22

Table 2: Accuracy of H-BD and T-BD BAYESDROID configurations

Results The results obtained for H-BD and T-BD are summarized in Table 2. Table 3 summarizes the findings reported by both H-BD and T-BD at the granularity of privacy items: the device number, identifier and location, while Table 5 provides a detailed description of the results across all benchmarks including those on which no leakages were detected. The warnings reported by the H-BD configuration are also publicly available for review.[5]

As Table 2 indicates, the H-BD variant is more accurate than the T-BD variant overall (F-measure of 0.98 vs. 0.73). As in the experiment described in Section 5.3, we further performed a two-tail McNemar test, considering 67 observations for each tool: 27 that correspond to true positives, 1 to the false positive due to H-BD and 39 to cases where no leakages were found.

We found that H-BD was accurate in 66 out of 67 cases, and T-DB was accurate in 54 out of 67 cases. Building the 2×2 contingency table and applying the two-tail McNemar test showed that the difference between the tools in accuracy is significant (with a p-value of 0.001 to reject the null hypothesis that the accuracy observations for both tools come from the same Bernoulli distribution). Moreover, H-BD has a lower number of crashes and lower runtime overhead, which confirms H2.

Discussion To give the reader a taste of the findings, we present in Figures 7–8 two examples of potential leakages that BAYESDROID (both the H-BD and the T-BD configurations) deemed legitimate. The instance in Figure 7 reflects the common scenario of obtaining the current (or last known) location, converting it into one or more addresses, and then releasing only the country or zip code. In the second instance, in Figure 8, the 64-bit Android ID — generated when the user first sets up the device — is read via a call to `Settings$Secure.getString(ANDROID_ID)`. At the release point, into the file system, only a prefix of the Android ID consisting of the first 12 digits is published.

As Table 3 makes apparent, the findings by H-BD are more complete: It detects 18 leakages (versus 8 reports by T-BD), with no false negative results and only one false positive. We attribute that to (i) the intrusive instru-

[5] See archive file researcher.ibm.com/researcher/files/us-otripp/realworldapps.zip.

App	Domain	Deep crawl?	H-BD			T-BD		
			number	dev. ID	location	number	dev. ID	location
atsoft.games.smgame	games/arcade	✓		✓	✓		✓	✓
com.antivirus	communication	✓		✓			✓	
com.appershopper.ios7lockscreen	**personalization**		✓	✓	✓			✓
com.bestcoolfungames.antsmasher	games/arcade	✓			✓			✓
com.bitfitlabs.fingerprint.lockscreen	**games/casual**			✓				
com.cleanmaster.mguard	tools	✓		✓			✓	
com.coolfish.cathairsalon	**games/casual**	✓		✓				
com.coolfish.snipershooting	**games/action**	✓		✓				
com.digisoft.TransparentScreen	entertainment	✓			✓			✓
com.g6677.android.cbaby	games/casual			✓				
com.g6677.android.chospital	games/casual			✓				
com.g6677.android.design	games/casual			✓				
com.g6677.android.pnailspa	games/casual			✓				
com.g6677.android.princesshs	games/casual			✓				
com.goldtouch.mako	news	✓		✓			✓	
15		8	1	13	4	0	4	4

Table 3: Warnings by the H-BD and T-BD BAYESDROID configurations on 15/54 top-popular mobile apps

source : private value

GeoCoder.getFromLocation(...) : [Lat: ..., Long: ..., Alt: ..., Bearing: ..., ..., **IL**]

sink : arguments

WebView.loadUrl(...) : http://*linux.appwiz.com*/profile /72/72_exitad.html?p1=RnVsbCtBbmRyb2lkK29uK0VtdWxhdG9y&p2=Y2RmMTUxMjRlYTRjN2FkNQ%3d%3d& ... LOCATION=**IL**& ... MOBILE_COUNTRY_CODE=&NETWORK=WIFI

Figure 7: Suppressed warning on ios7lockscreen

source : private value

Settings$Secure.getString (...) : **cdf15124ea4c7ad5**

sink : arguments

FileOutputStream.write (...) : <?xml version='1.0' encoding='utf−8' standalone='yes' ?><map><string name="openudid">**cdf15124ea4c**

Figure 8: Suppressed warning on fruitninjafree

mentation required for tag propagation, which can cause instabilities, and (ii) inability to track tags through native code, as discussed below.

The T-BD variant introduces significantly more instability than the H-BD variant, causing illegal application behaviors in 21 cases compared to only 12 under H-BD. We have investigated this large gap between the H-BD and T-BD configurations, including by decompiling the subject apps. Our analysis links the vast majority of illegal behaviors to limitations that TaintDroid casts on loading of third-party libraries. For this reason, certain functionality is not executed, also leading to exceptional app states, which both inhibit certain data leaks.[6]

A secondary reason why H-BD is able to detect more leakages, e.g. in the lockscreen app, is that this benchmark makes use of the `mobileCore` module,[7] which is a highly optimized and obfuscated library. We suspect that data-flow tracking breaks within this library, though we could not fully confirm this.

At the same time, the loss in accuracy due to heuristic identification of relevant values is negligible, as suggested by the discussion in Section 4.2. H-BD triggers only one false alarm, on ios7lockscreen, which is due to overlap between irrelevant values: extra information on the `Location` object returned by a call to `LocationManager.getLastKnownLocation(...)` and unrelated metadata passed into a `ContextWrapper.startService(...)` request. Finally, as expected, H-BD does not incur false negatives.

6 Related Work

As most of the research on privacy monitoring builds on the tainting approach, we survey related research mainly in this space. We also mention several specific studies in other areas.

[6] For a technical explanation, see forum comment by William Enck, the TaintDroid moderator, at https://groups.google.com/forum/#!topic/android-security-discuss/U1fteEX26bk.

[7] https://www.mobilecore.com/sdk/

Realtime Techniques The state-of-the-art system for realtime privacy monitoring is TaintDroid [4]. TaintDroid features tolerable runtime overhead of about 10%, and can track taint flow not only through variables and methods but also through files and messages passed between apps. TaintDroid has been used, extended and customized by several follow-up research projects. Jung et al. [10] enhance TaintDroid to track additional sources (including contacts, camera, microphone, etc). They used the enhanced version in a field study, which revealed 129 of the 223 apps they studied as vulnerable. 30 out of 257 alarms were judged as false positives. The Kynoid system [20] extends TaintDroid with user-defined security policies, which include e.g. temporal constraints on data processing as well as restrictions on destinations to which data is released.

The main difference between BAYESDROID and the approaches above, which all apply information-flow tracking, is that BAYESDROID exercises "fuzzy" reasoning, in the form of statistical classification, rather than enforcing a clear-cut criterion. As part of this, BAYESDROID factors into the privacy judgment the data values flowing into the sink statement, which provides additional evidence beyond data flow.

Quantitative Approaches Different approaches have been proposed for quantitative information-flow analysis, all unified by the observation that data leakage is a quantitative rather than boolean judgment. McCamant and Ernst [13] present an offline dynamic analysis that measures the amount of secret information that can be inferred from a program's outputs, where the text of the program is considered public. Their approach relies on taint analysis at the bit level. Newsome et al. [14] develop complementary techniques to bound a program's *channel capacity* using decision procedures (SAT and #SAT solvers). They apply these techniques to the problem of false positives in dynamic taint analysis. Backes et al. [1] measure leakage in terms of indistinguishability, or equivalence, between outputs due to different secret artifacts. Their characterization of equivalence relations builds on the information-theoretic notion of entropy. Budi et al. [2] propose *kb*-anonymity, a model inspired by *k*-anonymity that replaces certain information in the original data for privacy preservation, but beyond that also ensures that the replaced data does not lead to divergent program behaviors.

While these proposals have all been shown useful, none of these approaches has been shown to be efficient enough to meet realtime constraints. The algorithmic complexity of computing the information-theoretic measures introduced by these works seriously limits their applicability in a realtime setting. Our approach, instead, enables a quantitative/probabilistic mode of reasoning that is simultaneously lightweight, and therefore acceptable for online monitoring, by focusing on relevant features that are efficiently computable.

Techniques for Protecting Web Applications There exist numerous static and dynamic approaches for preventing attacks on web applications, e.g., [23, 22, 7]. Most relevant to our work are Sekar's taint-inference technique for deducing taint propagation by comparing inputs and outputs of a protected server-side application [21] and a similar browser-resident technique developed in a subsequent study [16]. While BAYESDROID shares ideas with these approaches, it is explicitly designed for mobile devices and applications. Curtsinger et al. [3] apply a Bayesian classifier to identify JavaScript syntax elements that are highly predictive of malware. The proposed system, ZOZZLE, analyzes the application's code statically, while BAYESDROID operates dynamically and focuses on data values.

7 Conclusion and Future Work

In this paper, we articulated the problem of privacy enforcement in mobile systems as a classification problem. We explored an alternative to the traditional approach of information-flow tracking, based on statistical reasoning, which addresses more effectively the inherent fuzziness in leakage judgements. We have instantiated our approach as the BAYESDROID system. Our experimental data establishes the high accuracy of BAYESDROID as well as its applicability to real-world mobile apps.

Moving forward, we have two main objectives. The first is to extend BAYESDROID with additional feature types. Specifically, we would like to account for (i) sink properties, such as file access modes (private vs public), the target URL of HTTP communication (same domain or third party), etc; as well as (ii) the history of privacy-relevant API invocations up to the release point (checking e.g. if/which declassification operations were invoked). Our second objective is to optimize our flow-based method for detecting relevant values (see Section 3.1) by applying (offline) static taint analysis to the subject program, e.g. using the FlowDroid tool [6].

References

[1] M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *S&P*, pages 141–153, 2009.

[2] A. Budi, D. Lo, L. Jiang, and Lucia. kb-anonymity: a model for anonymized behaviour-preserving test and debugging data. In *PLDI*, pages 447–457, 2011.

[3] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security*, pages 33–48, 2011.

[4] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 1–6, 2010.

[5] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, pages 21–21, 2011.

[6] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, 2014.

[7] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA*, pages 177–187, 2011.

[8] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. Flow permissions for android. In *ASE*, pages 652–657, 2013.

[9] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*, pages 639–652, 2011.

[10] J. Jung, S. Han, and D. Wetherall. Short paper: enhancing mobile application permissions with runtime feedback and constraints. In *SPSM*, pages 45–50, 2012.

[11] B. Livshits and J. Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *USENIX Security*, pages 113–130, 2013.

[12] G. Lowe. Quantifying information flow. In *CSFW*, pages 18–31, 2002.

[13] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, pages 193–205, 2008.

[14] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS*, pages 73–85, 2009.

[15] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[16] Riccardo Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *ASIACCS*, pages 5–5, 2012.

[17] J. Piskorski and M. Sydow. String distance metrics for reference matching and search query correction. In *BIS*, pages 353–365, 2007.

[18] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODAPSY*, pages 209–220, 2013.

[19] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.

[20] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *WISTP*, pages 208–223, 2012.

[21] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[22] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

[23] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[24] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[25] Bernard L Welch. The generalization of student's problem when several different population variances are involved. *Biometrika*, 34(1–2):28–35, 1947.

[26] D. Wetherall, D. Choffnes, B. Greenstein, S. Han, P. Hornyack, J. Jung, S. Schechter, and X. Wang. Privacy revelations for web and mobile apps. In *HotOS*, pages 21–21, 2011.

[27] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, pages 1043–1054, 2013.

Benchmark	Algorithm	TPs	FPs	FNs
ActivityCommunication1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle2	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ActivityLifecycle4	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Library2	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Obfuscation1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
PrivateDataLeak3	BAYESDROID	1	1	0
	TaintDroid	1	1	0
AnonymousClass1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ArrayAccess1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ArrayAccess2	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
HashMapAccess1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
Button1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Button3	BAYESDROID	2	0	0
	TaintDroid	2	0	0
Ordering1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
RegisterGlobal1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
DirectLeak1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
FieldSensitivity2	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
FieldSensitivity3	BAYESDROID	**1**	**0**	**0**
	TaintDroid	**1**	**0**	**0**
FieldSensitivity4	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ImplicitFlow1	BAYESDROID	**0**	**0**	**2**
	TaintDroid	**2**	**0**	**0**
InheritedObjects1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ListAccess1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
LocationLeak1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
LocationLeak2	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
Loop1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Loop2	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ApplicationLifecycle1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ApplicationLifecycle3	BAYESDROID	1	0	0
	TaintDroid	1	0	0
MethodOverride1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
ObjectSensitivity1	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**1**	**0**
ObjectSensitivity2	BAYESDROID	**0**	**0**	**0**
	TaintDroid	**0**	**2**	**0**
Reflection1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Reflection2	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Reflection3	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Reflection4	BAYESDROID	1	0	0
	TaintDroid	1	0	0
SourceCodeSpecific1	BAYESDROID	5	0	0
	TaintDroid	5	0	0
StaticInitialization1	BAYESDROID	1	0	0
	TaintDroid	1	0	0
Total	BAYESDROID	29	1	2
	TaintDroid	31	17	0

Table 4: Detailed summary of the results of the H1 experiment described in Section 5.3

A Overhead Measurement: Methodology

To complete the description in Section 5.4, we now detail the methodology governing our overhead measurements. The behavior of the benchmark app is governed by two user-controlled values: (i) the length ℓ of the source/sink data-flow path (which is proportional to the number of loop iterations) and (ii) the number m of values reachable from sink arguments.

Based on our actual benchmarks, as well as data reported in past studies [23], we defined the ranges $1 \leq \ell \leq 19$ and $1 \leq m \leq 13 = \Sigma_{n=0}^{2} 3^n$. We then ran the parametric app atop a "hybrid" configuration of BAYESDROID that simultaneously propagates tags and treats all the values flowing into a sink as relevant. For each value of ℓ, we executed the app 51 times, picking a value from the range $[0,2]$ for n uniformly at random in each of the 51 runs. We then computed the average overhead over the runs, excluding the first (cold) run to remove unrelated initialization costs. The stacked columns in Figure 6 each correspond to a unique value of ℓ.

B Detailed Results

Table 4 summarizes the results of the H1 experiment described in Section 5.3. For each of the benchmarks, it specifies the number of true-positive, false-positive and false-negative findings for the compared tools, BAYESDROID and TaintDroid. The benchmarks on which the tools differ are highlighted for convenience.

Similarly, Table 5 summarizes the results of the H2 experiment described in Section 5.4. The first two columns of Table 5 list the applications and their respective domain, and the third column denotes whether crawling was exhaustive. Then, the number of crashes, true-positive, false-positive and false-negative findings are reported for both the H-BD and the T-BD variants of BAYESDROID.

In Section 5.4, we describe an experiment designed to evaluate our Bayesian analysis in "pure" form, i.e. without the support of information-flow tracking to detect relevant values. To make our description of this experiment complete, we include Table 5, which provides a detailed summary of the results of this experiment across all benchmarks (including ones on which no leakages were detected). For comparability between the H-BD and T-BD configurations, we count different dynamic reports involving the same pair of source/sink APIs as a single leakage instance.

App	Domain	Deep crawl?	H-BD				T-BD			
			Crashes	TPs	FPs	FNs	Crashes	TPs	FPs	FNs
air.au.com.metro.DumbWaysToDie	games/casual		✓	0	0	0	✓	0	0	0
at.nerbrothers.SuperJump	games/arcade			0	0	0	✓	0	0	0
atsoft.games.smgame	**games/arcade**	✓		**4**	**0**	**0**		**4**	**0**	**0**
com.antivirus	**communication**	✓		**1**	**0**	**0**		**1**	**0**	**0**
com.appershopper.ios7lockscreen	**personalization**			**5**	**1**	**0**		**3**	**0**	**3**
com.applicaster.il.hotvod	entertainment	✓		0	0	0		0	0	0
com.appstar.callrecorder	tools			0	0	0		0	0	0
com.awesomecargames.mountainclimbrace_1	games/racing		✓	0	0	0	✓	0	0	0
com.bestcoolfungames.antsmasher	**games/arcade**	✓		**2**	**0**	**0**		**2**	**0**	**0**
com.bigduckgames.flow	games/puzzles			0	0	0	✓	0	0	0
com.bitfitlabs.fingerprint.lockscreen	**games/casual**			**2**	**0**	**0**		**0**	**0**	**1**
com.channel2.mobile.ui	news	✓		0	0	0		0	0	0
com.chillingo.parkingmaniafree.android.rowgplay	games/racing		✓	0	0	0	✓	0	0	0
com.cleanmaster.mguard	**tools**	✓		**1**	**0**	**0**		**1**	**0**	**0**
com.coolfish.cathairsalon	**games/casual**	✓		**2**	**0**	**0**	✓	**0**	**0**	**1**
com.coolfish.snipershooting	**games/action**	✓		**2**	**0**	**0**	✓	**0**	**0**	**1**
com.cube.gdpc.isr	health & fitness		✓	0	0	0	✓	0	0	0
com.cyworld.camera	photography			0	0	0		0	0	0
com.devuni.flashlight	tools	✓		0	0	0		0	0	0
com.digisoft.TransparentScreen	**entertainment**	✓		**2**	**0**	**0**		**2**	**0**	**0**
com.domobile.applock	tools	✓		0	0	0		0	0	0
com.dropbox.android	productivity	✓		0	0	0	✓	0	0	0
com.ea.game.fifa14_row	games/sports			0	0	0	✓	0	0	0
com.ebay.mobile	shopping			0	0	0		0	0	0
com.facebook.katana	social	✓		0	0	0		0	0	0
com.facebook.orca	communication			0	0	0		0	0	0
com.g6677.android.cbaby	**games/casual**			**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.chospital	**games/casual**		✓	**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.design	**games/casual**		✓	**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.pnailspa	**games/casual**		✓	**1**	**0**	**0**	✓	**0**	**0**	**1**
com.g6677.android.princesshs	**games/casual**			**1**	**0**	**0**	✓	**0**	**0**	**1**
com.gameclassic.towerblock	games/puzzles	✓		0	0	0	✓	0	0	0
com.gameloft.android.ANMP.GloftDMHM	games/casual			0	0	0		0	0	0
com.game.fruitlegendsaga	games/puzzles			0	0	0		0	0	0
com.gau.go.launcherex	personalization			0	0	0		0	0	0
com.glu.deerhunt2	games/arcade		✓	0	0	0	✓	0	0	0
com.goldtouch.mako	**news**	✓		**1**	**0**	**0**		**1**	**0**	**0**
com.goldtouch.ynet	news	✓		0	0	0		0	0	0
com.google.android.apps.docs	productivity			0	0	0		0	0	0
com.google.android.apps.translate	tools			0	0	0		0	0	0
com.google.android.youtube	media & video			0	0	0		0	0	0
com.google.earth	travel & local		✓	0	0	0	✓	0	0	0
com.halfbrick.fruitninjafree	games/arcade			0	0	0		0	0	0
com.halfbrick.jetpackjoyride	games/arcade	✓		0	0	0		0	0	0
com.icloudzone.AsphaltMoto2	games/racing			0	0	0	✓	0	0	0
com.ideomobile.hapoalim	finance			0	0	0		0	0	0
com.imangi.templerun2	games/arcade		✓	0	0	0	✓	0	0	0
com.kiloo.subwaysurf	games/arcade		✓	0	0	0	✓	0	0	0
com.king.candycrushsaga	games/arcade		✓	0	0	0	✓	0	0	0
com.sgiggle.production	social			0	0	0		0	0	0
com.skype.raider	communication			0	0	0		0	0	0
com.UBI.A90.WW	games/arcade			0	0	0		0	0	0
com.viber.voip	communication			0	0	0		0	0	0
com.whatsapp	communication			0	0	0		0	0	0
Total		17	12	27	1	0	22	14	0	10

Table 5: Detailed summary of the results of the H2 experiment described in Section 5.4

The Long "Taile" of Typosquatting Domain Names

Janos Szurdi[◇] Balazs Kocso[*] Gabor Cseh[*]
Jonathan Spring[◇] Mark Felegyhazi[*] Chris Kanich[†]

[◇]*Carnegie Mellon University* [*]*Budapest University of Technology and Economics*
[†]*University of Illinois at Chicago*

Abstract

Typosquatting is a speculative behavior that leverages Internet naming and governance practices to extract profit from users' misspellings and typing errors. Simple and inexpensive domain registration motivates speculators to register domain names in bulk to profit from display advertisements, to redirect traffic to third party pages, to deploy phishing sites, or to serve malware. While previous research has focused on typosquatting domains which target popular websites, speculators also appear to be typosquatting on the "long tail" of the popularity distribution: millions of registered domain names appear to be potential typos of other site names, and only 6.8% target the 10,000 most popular `.com` domains.

Investigating the entire distribution can give a more complete understanding of the typosquatting phenomenon. In this paper, we perform a comprehensive study of typosquatting domain registrations within the `.com` TLD. Our methodology helps us to significantly improve upon existing solutions in identifying typosquatting domains and their monetization strategies, especially for less popular targets. We find that about half of the possible typo domains identified by lexical analysis are truly typo domains. From our zone file analysis, we estimate that 20% of the total number of `.com` domain registrations are true typo domains and their number is increasing with the expansion of the `.com` domain space. This large number of typo registrations motivates us to review intervention attempts and implement efficient user-side mitigation tools to diminish the financial benefit of typosquatting to miscreants.

1 Introduction

Thousands of new domain names are registered daily that at first glance do not have completely legitimate uses: some contain random characters (possibly used by miscreants [23]), are a composite of two completely unrelated words (possibly used in spam [17]), contain keywords of highly-visible recent events (ex. `hillaryclingon.com` for political phishing in 2008 [28]) or are similar to other, typically well-known, domain names (ex. `twtter.com` [27, 32]). Domain purchasers use this final technique, often called "typosquatting," to capitalize on other domain names' popularity and user mistakes to drive traffic to their websites.

Many old and new domain names alike do not ever show up in search engines, spam traps, or malicious URL blacklists, yet still maintain a web server hosting some form of content. However, maintaining the domain registration, DNS, and web server expends resources, even if these domain registrations do not serve an obvious purpose. Investigating the purpose of domain registrations in the "long tail" of the popularity distribution can help us better understand these enterprises and their relationship to speculative and malicious online activities. In this paper, we specifically consider the hypothesis that typosquatting is a reason for many of these registrations, and scrutinize different methods for committing malice or monetizing this behavior.

In the Internet economy, monetizing on user intent has been a very profitable business strategy: search display advertising is effective because relevant ads can be shown based on user search queries. DNS is similar, as domain registrations provide ample opportunities for monetization through direct user navigation rather than search. Domain name front running, domain tasting and typosquatting domain names can all monetize this phenomenon. [1] [12] According to [22], domain tasting was nearly eliminated in the generic TLDs by the 2009 policy changes by ICANN. In addition, [12] reports that the

[1] *Domain name front running* is when registrars register domains that users have been looking for in order to monetize on their registration potential. *Domain tasting* is speculative behavior abusing the five-day grace period after domain registrations in some TLDs. This liberal registration policy gave refunds within a few days if the registrant wanted, however this policy resulted in short domain registrations en masse. ICANN has since changed policy, limiting the behavior [12, 22].

anecdotes about domain name front running by major registrars do not seem to hold. But typosquatting, the most prevalent speculative domain name registration behavior to date, continues apace.

Typosquatting wastes users' time and no doubt annoys them as well. As we show in Section 4.5, less than two percent of all domains we identify as "typo domains" redirect the user to the targeted domain, and the lion's share instead serve advertisements which previous research has shown to be profitable. [16, 26] These ad-filled pages give no clear indication to the user that they have typed the domain incorrectly; without a descriptive error, the user may abandon their task rather than double check their spelling. By monetizing these pages with advertisements, the typosquatter does a disservice both to the user and the victim web site. Protecting users from typosquatters can lessen the damage as well as disincentivize typosquatting by decreasing the squatters' profits.

If a typosquatter hosts a site that impersonates the legitimate brandholder it is certainly malicious and in some jurisdictions illegal. Such overt violations have been mitigated via legislation in the US and policy by ICANN [15, 21, 30]. For example, Facebook recently extracted a $2.8 million judgement against typosquatters impersonating their website; this successful litigation should serve as a strong deterrent against this form of malicious typosquatting against entities with the resources to litigate [18]. Several reports by commercial security teams have cited typosquatting domains' use in malicious campaigns for quiz scams [8], spam survey sites [37], in an SMS micro-payment scam [14], offering deceptive downloads or serving adult content [25], or in a bait-and-switch scam offering illegal music downloads [29]. However, until this paper, evidence regarding the extent of malicious typosquatting problems has not been available.

Typosquatting has been studied in depth in related work. In his first paper, Edelman points to the typosquatting phenomenon and discusses possible incentives for both squatters and defenders [15]. Wang *et al.* include a typopatrol service in their Strider security framework that focuses on generating typo domains for popular domains and protect visitors from offending content [35]. Moore and Edelman revisit the problem in [26] pursuing a more thorough study of the original thesis of Edelman. They explore various monetization methods and suggest intervention options. They pessimistically conclude that the best intervention options are hampered by misaligned incentives of the participants. Banerjee *et al.* [10] make another attempt to design a typosquatting categorization tool. Their method works well for a small set of sample domain names. These analyses have focused on active measurement of typosquatting sites which target the most popular domains – considering no more than 3,264 unique `.com` domain names. However, we find that no more than 4.9% of all lexicographically similar name registrations target these popular domains. While typos for the most popular domains likely account for a significant amount of typo traffic, it is unclear whether the long tail also supports a significant amount of typo traffic.

Here we present a systematic study of domain name registrations focusing on typosquatting perpetrated against the long tail of the popularity distribution. We design a set of algorithms that can effectively identify typosquatting domains and categorize the monetization method of its owner. We also design and implement tools to improve user experience by allowing them to reach their intended destination. Although various user tools exist in the wild, most are inaccurate and focus only on a limited set of targeted domains. Our typo identification algorithms combined with the user protection tools provide improved protection against being misled by typosquatting, even when it is perpetrated against less popular sites.

Section 2 provides background on typosquatting and the most common tricks used by typosquatters. Section 3 presents our data collection methodology and describes our typo categorization framework. Section 4 presents a characterization of the extent, purpose, trends, and malice involved in the perpetration of typosquatting. We present mitigation tools and intervention options in Section 5. Section 6 concludes.

2 Background

Popularity attracts speculation, and typosquatting is a showcase of this observation in the Internet ecosystem. Typosquatting maintains its popularity even in the face of the continuous effort to diminish its impact. In this section, we present a general overview of typosquatting and discuss efforts to protect legitimate domain owners from speculation.

2.1 Typo techniques and monetization

Typosquatters register domain names that are similar to those used by other websites in hope of attracting traffic due to user mistakes. The most frequent occurrences of mistyping are those that involve a one-character distance, also called the Damerau-Levenshtein (DL) distance one, from the correct spelling both in free text [13] and in case of domain names [10][2]. In this paper, we focus on typosquatting domains of Damerau-Levenshtein distance one (DL-1) that are generated using the most common operations: addition, deletion, substitution of one character, transposition of neighboring characters [13]. We extend this to include deletion of the period before the "www"

[2] Although some researchers have found that for longer original domains a small number of typosquatting domain names with larger DL distances exist [26].

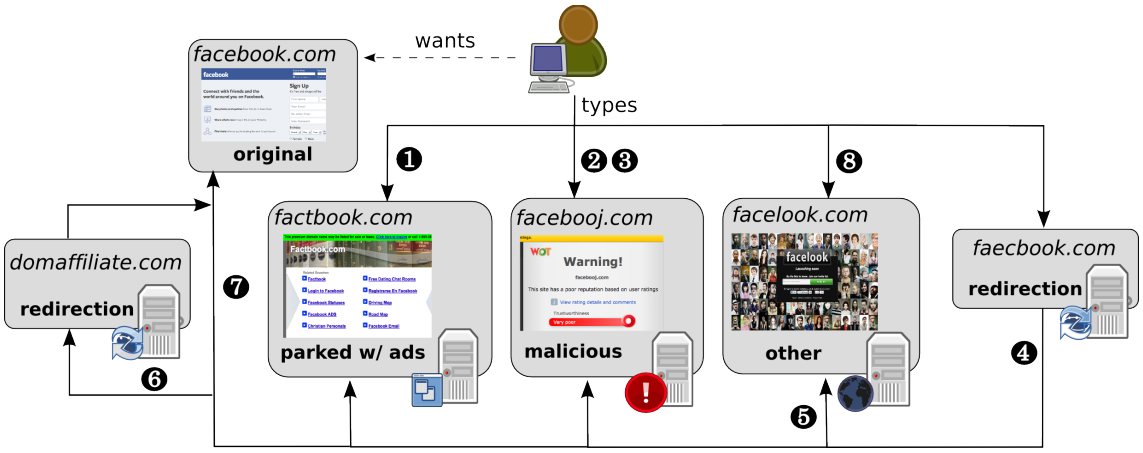


Figure 1: The typosquatting ecosystem with various monetization techniques.

commonly prepended to web server domain names [26]. We note that a special case of DL-1, called fat finger distance (FF distance), is considered when the mistyping occurs with letters that are adjacent on a US English keyboard. The rationale of this metric is that users are more likely to mistype letters in close proximity.

Typosquatters use various techniques to monetize their domain name registrations. The typosquatting domain can be *parked* and serve third-party advertisements to monetize the incoming traffic (❶ on Figure 1). The domain can also be set up to impersonate the intended domain for instance to host a phishing page [33] (❷), serve malware (❸), or perpetrate some other scam on the user [14, 37]. Many monetization techniques can also involve redirection to another domain (❹), the *landing domain*, that might employ the previously mentioned techniques. Speculators can also redirect visitors to *competitor domains* (❺) causing a direct loss to the owner of the original domain. Conversely, the typodomain owner can redirect traffic to the intended site, and monetize this traffic via *affiliate marketing* (❻). The original domain owner can also perform *defensive registrations* of typos for their main domain name and set up the redirections themselves (❼). Finally, in some cases, the typo domain owner can serve content that is unrelated to the original domain (❽).

2.2 Intervention attempts

Typosquatting exists within a legal and moral gray area; consequently, intervention has traditionally been weak to reduce the effect of typosquatting. ICANN provides the Uniform Domain-Name Dispute-Resolution Policy (UDRP) to mediate domain registration disputes for a relatively small filing fee. Unfortunately, cheap domain registration allows for mass typo-domain registrations and this gives a significant advantage to speculators. Against mass registrations of typo-domains UDRP mitigation becomes infeasible. Companies have initiated legal procedures in cases where cybersquatting and trademark infringement was applicable (see for example [32] on a recent court order against `twtter.com` and `wikapedia.com`, and a more recent court order against typosquatters of `facebook.com` [31]). The Anti-cybersquatting Consumer Protection Act (ACPA) (15 USC §1125(d)) offers legal protection to push such cases to court.

Policy intervention is more effective when targeting the registration process either at a national scale for specific TLDs or on a registrar level [24]. One can also mount an effective defense by targeting the monetization infrastructure [23, 24]. Unfortunately, the agility of domain speculators in registering new domains and the difficulty of determining their ill intent makes this a difficult prospect.

There have been some efforts to provide technical tools to mitigate typosquatting, notably the Microsoft Strider Typopatrol system which protects trademarks and childrens' sites [35]. At the user level, the OpenDNS has a typo correction feature which corrects major TLD misspellings [27] and the Mozilla URLFixer Firefox plugin [6] can suggest corrections to typed URLs. A common property of these solutions is that they only cover a relatively small set of typos, typically those that target the most popular domain names. As we show in Section 5.3, our mitigation solution is based on an extensive set of investigated domain names and hence provides significantly better coverage to detect typosquatting. Moreover, our extended set of detection features allows for more accurate detection of typosquatting than solutions in previous work.

3 Methodology

This section presents our data collection and domain categorization framework in detail as illustrated it in Figure 2.

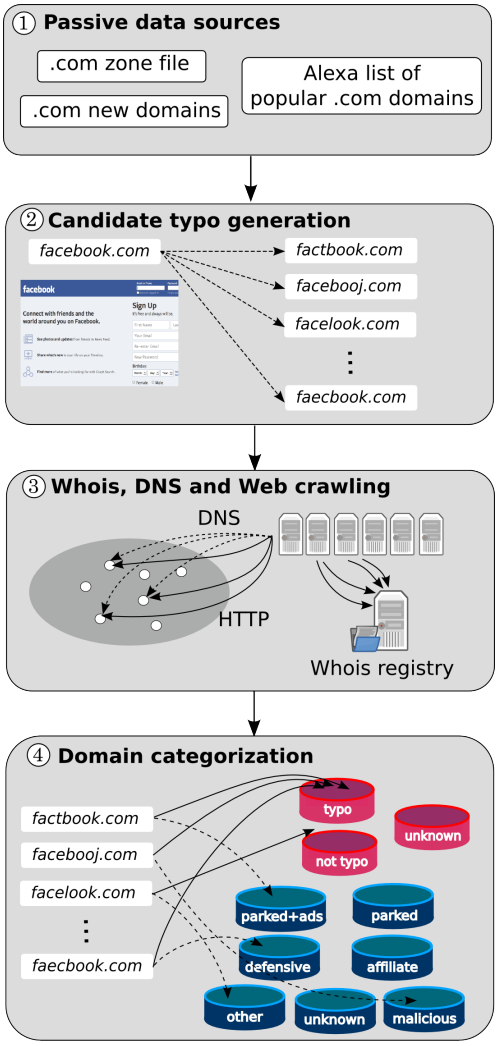


Figure 2: The data collection and typo categorization framework. The framework uses (①) large domain lists (zone file, Alexa popular domains list), (②) derives candidate typos based on lexical features and registration data in the zone file, (③) acquires additional information using active crawlers (Whois, DNS, Web), and finally (④) decides about typo domains and assigns them into typosquatting categories.

Terminology. Throughout this paper, we will refer to domains available for direct registration under a public suffix as *registered domains*, for instance *example.com* or *example.co.uk*. Generated typo domains, or *gtypos*, are domain names which are lexically similar (e.g. at DL-1) to some set of target domains. Candidate typo domains, or *ctypos*, are the subset of registered domains within the *gtypo* set which have been registered. Below we describe both how we select the target set and how we generate the gtypo set.

3.1 Data sources and scope

`.com` zone file. We leverage a variety of data sources to infer the prevalence of typosquatting in domain registrations. Our primary source is the `.com` zone file, which contains records of every domain registered within that TLD. As a popular generic domain name, the `.com` zone file contains millions of registered domain names `.com` and is available to researchers making it an ideal candidate for a representative investigation of typosquatting. Our comprehensive study is based on the March 15, 2013 version of the zone file provided by Verisign Inc containing approximately 106 million domain names. For trend analysis we collected the daily newly added and deleted domains from the zone file from October 01, 2012 to February 20, 2014.

Alexa list. The Alexa list of the top 1 million sites from March 15, 2013 serves as a benchmark for popularity [1], out of which 523,960 domains belong to the `.com` TLD, with 488,113 unique registered domains five characters long or more. For our study, we split the Alexa list into three categories: *Alexa top* containing domains ranked higher than 10,000, *Alexa mid* containing domains ranked 10,000-250,000, and *Alexa tail* containing the remaining `.com` domains ranked below 250,000. While Alexa cautions that rankings below 100,000 are not statistically significant, we are not concerned with exact comparative ranking or traffic counts for these domains but consider the Alexa list rather as a rough indicator of popularity. We also collected the Alexa top 1 million for the October 01, 2012 to February 20, 2014 period for trend analysis.

Domain blacklists. To shed light on the malicious use of typo domains, we check the typo domains from the `.com` zone file against twelve different domain name blacklists. The black lists come from abuse.ch's list of Zeus and SpyEye servers, malwaredomainlist.com, malwaredomains.com, malwarepatrol.com, Google Safe Browsing, and a commonly used commercial list. We also derive lists of malicious domains from recorded requests to DNS-based black lists (DNSBL). This method does not capture the complete list, but rather only includes domains actively marked as malicious and looked up by users during the collection time frame.

3.2 Generating candidate typos

We generated a list of all possible typo domains using the most common typo operations: addition (*add*), deletion (*del*), substitution of one character (*sub*), transposition of neighboring characters (*tra*), and supplement this set with a "." deletion operation specific to "www." domain names (e.g. a user typed (*wwwexample.com*). We define this list as the "generated typo" or *gtypo* list. The subset of the gtypo list which was registered within the `.com` TLD

includes approximately 4.7 million domains, which we refer to as "candidate typos" or *ctypos*.

3.3 Typosquatting definitions

To define the scope of our work, we provide a concise definition of typosquatting.

Definition 1 *A candidate typo domain is called a* typosquatting domain *if (i) it was registered to benefit from traffic intended for a target domain (ii) that is the property of a different entity.*

It is important that both conditions have to be met simultaneously. Typosquatting domain names are registered with the parasitic intent to reap the mistyped traffic of popular domains belonging to someone else. This includes parked domains serving ads, phishing domains, known malicious domains, typo domains redirecting to unrelated content and affiliate marketing. Arguably, these conditions cannot always be checked with confidence, for example ownership information could be disguised[3].

According to our definition, parked domains that do not serve ads are excluded from our definition of typosquatting, because they are not making any visible profit from parking. We still consider them as typos until it becomes clear if they are performing typosquatting on the target or serving unrelated content. Candidate typo domains that are defensively registered by the original domain owner are also excluded from typosquatting, because the owner of the typo domain and the original domain are the same. Although defensive typo registrations cannot be considered as typosquatting, they are born as an unwanted consequence of typosquatting.

We define *true typo* domains as follows.

Definition 2 *We call the union of typosquatting domains, parked domains not serving ads and defensive registrations the* true typo *domain set.*

Finally, all candidate typos that are at DL-1 from an original domain yet have unrelated content are considered as incidental registrations, although they can surely benefit from the lexical proximity[4].

3.4 Active crawling

We developed a set of active crawlers to collect additional information about the ctypo domains.

[3]For example, the name servers `*.aexp.com` of `americanexpressl.com` belong to American Express Inc., but that is the only indicator of ownership. This can only be marked using manual inspection.

[4]Here we face another uncertainty presented by scam pages that generate legitimately looking random content. We observed several such cases for suspiciously looking webshops. We make a conservative assessment and categorize them as other (O) in spite of their questionable content

Whois crawler. First, we collect registration data from the WHOIS global database. We restrict our crawler to the thin whois information as provided by Verisign Inc. for the `.com` domains. From the thin whois record, we use the registrar and registration date information.

DNS crawler. We collect DNS data to explore the background infrastructure serving these domains. Our crawler queries separately for A, AAAA, NS, MX, TXT, CNAME, and SOA records for each domain. The crawler then tests for random strings under the registered domain to infer whether wildcarding is present. Wildcarding is the practice when a name server resolves any subdomain under the domain belonging to its authority in the DNS hierarchy.

Web crawler. We use a web crawler to obtain the rendered DOM of each page, along with any automatic redirections that take place during the page load. This crawler uses the PhantomJS WebKit automation framework to provide high volume, full fidelity web crawling with javascript execution, cookie storage, and page rendering capabilities [20]. The crawler follows JavaScript redirections even when they may be obfuscated or contained in child iframes; it then reports the method of redirection and the destination for intermediate and final redirections. We also collect rendered screenshots of a subset of pages for manual inspection.

3.5 Clustering and categorization

Clustering. We group domains together according to various attributes obtained from available datasets and active analysis. Our goal with this clustering is twofold: to identify typo domains that might have been registered for the same purpose and to point to infrastructure elements that host a large number of typo domains. First, we identify domain sets that are at DL-1 distance from each other, forming a cluster of *typo neighbors*.

Understanding the infrastructure support and the content of the typo domains is required to make an informed decision about their real purpose. To characterize the infrastructure support for typosquatting, we cluster the candidate typo domains based on their registration and hosting information. In particular, we identify the major registrars and name servers (NSs) that host candidate typo domains.

Domain features. We derive a feature set including lexical, infrastructure and content features of the candidate typos as shown in Table 5 in Appendix A. We selected the features after carefully considering related work, collecting 40+ features in various attribute categories, and focusing only on relevant ones. To assess the efficiency of the selected feature set, we perform a systematic evaluation based on manual sampling in Section 4.1

and we use the results of this evaluation as a benchmark.[5]

Among the chosen features, domain length is a key indicator for typosquatting behavior as longer ctypo domains are more likely to indeed typosquat on the original domain they are close to [26]. Intuitively, the Alexa rank of the original domain indicates that more popular domains are more likely a target of typosquatting. Based on the zone file, we are able to observe the ratio of ctypo domains versus all domain names on a given NS and we deem hosting a large of proportion of potential typo domains suspicious for an NS. Similarly, if the registered domain of the NS contains keywords indicating parking behavior, then ctypo domains hosted on this NS are more likely to belong to typosquatting domains. NXDOMAIN wildcarding is used by major parking service providers to serve ads for web requests regardless of the subdomain. It has been shown that NXDOMAIN wildcarding is a precursor of suspicious behavior and quite often indicates parked typosquatting domains [7, 36]. Thus, we also consider it an indicator for typosquatting when the page content matches some collected parking keywords[6]. Finally, several redirections usually imply suspicious behavior, and we deem them important if the redirection targets a registered domain different from the typo domain and the target domain. The features we selected resulted in a significant improvement over existing methods in identifying typosquatting domains across the whole range of `.com` domains. We leave a more complex feature set selection and parameter calibration using machine learning techniques as future work.

Categorization. Using these features, we attribute typosquatting to candidate typo (ctypo) domains by assigning the tag *typosquatting (T)*, *not typosquatting (NT)* or *unknown (U)*. Unknown is typically used when the domain returns an HTTP or DNS error which prevents successfully downloading the page. We also tag the usage type of the typosquatting domains according to the monetization categories presented in Figure 1. We also present the novel approach of categorizing domains based on their monetization strategy. Hence, we tag ctypo domains which do not redirect the user to the target site as *parked (P)* without ads (not on Figure 1), *parked serving ads (PA)* (❶ on Figure 1), employing a *phishing (PH)* scam (❷), or serving *malware (M)* (❸). When redirection is used, then the ctypo domain can be tagged as *defensive (D)* registration (❼), defensive registration using *affiliate (A)* marketing (❻) in addition to the previously mentioned categories. When a ctypo domain redirects to another domain, then we tag it as *other (O)* (❹, ❺) no matter if it is a competitor or a completely unrelated site[7]. Finally, we mark all uncategorized domains as *unknown (U)*, a set that typically contains unreachable domains.

3.6 Checking Maliciousness

To analyze how the typo domains are used, 12 black lists are checked for an indication that the domains are malicious. To check a black list, we look for anything that was on that list during the first quarter of 2013. A "match" is a second-level domain match, since this is the relevant typo label.

To perform a check, a superset of all the domains for Q1 2013 per list was made, and the typo and Alexa domains were compared against that superset. For Google Safe Browsing, due to Google's technical constraints, the each set of domains was checked using the provided python client against data for May 1, 2011 to July 31, 2013. The results are presented in subsection 4.6.

4 Analysis

In this section, our goal is to characterize the current state of typosquatting. For this purpose, we use the `.com` zone file as the most popular and versatile TLD for domain registrations.

4.1 Typosquatting distribution

Experts believe that most newly registered domains are speculative or malicious. Paul Vixie posits that "most new domain names are malicious" [34]. The subset of registered typo domains from the generated typo domains is widely accepted as true typo domains ([26, 35]), and [26] has shown that this assertion mostly holds for the top 3,264 `.com` domains in the Alexa ranking.

We believe, however, that this assertion does not necessarily hold if we extend our scope to less popular domains. In order to investigate this possibility, we first perform a manual sampling from various sets of the `.com` zone file to systematically control the accuracy of typosquatting identification and also to provide a credible ground truth for investigation. We conduct a manual inspection of four thousand domain names because the typosquatting definitions in the academic literature [26, 35] are very crude. Moreover, we present our mitigation tool analysis in Section 5, and in so doing also discuss the limitations of existing defense tools that typically only focus on correcting typos for a limited set of popular domain names.

[5] Manually generated datasets are widely used as indicators for malicious behavior; for example, the PhishTank phishing list is a major component of SURBL, the leading domain blacklist. [2].

[6] Here, we improve on the techniques used by [7] and [19] to find parking services and parked domains

[7] Determining domain competitors is beyond the scope of this work; we summarized redirections to third-party domains independently of the typosquatter's intent. While these redirections might simply be to other parked sites, any redirection away from the original site is a traffic loss for the original domain owner.

We first take a sample of 1000 ctypo domains randomly with uniform distribution from the Alexa top domain list to match the sampling methodology of [26]. We then complete this with three additional samples of 1000 ctypo domains each derived from the `.com` zone and the Alexa domain list. Our four sample sets are thus the following: ctypos of the the Alexa top/mid/tail domains (recall their description from Section 3.1) and ctypos of a random sample taken over the whole `.com` zone file. With these multiple sets, our goal is to check whether the conclusions from prior work regarding the frequency of typosquatting hold for less popular domains.

Typosquatting domains are notoriously difficult to identify. In several cases, only a careful investigation shows the potentially speculative behavior. We performed manual verification to establish a ground truth for identifying typosquatting domains. Clearly, manual classification is not perfect, but it allowed us to go in depth at domains that were ambiguous. In manual classification, we go beyond simple rules, like identifying simple one-hop defensive redirections and consider the environment, like the owner of name servers (`ns*.aexp.com` indeed belongs to American Express Inc) or potential relation between brands (Oldnavy is a subsidiary of GAP and thus `oldnavy.com` redirects to `oldnavy.gap.com`). We could further establish a ground truth based on crowdsourcing typosquatting identification. This would remove the bias introduced by the mindset of the authors, yet it could introduce significant inaccuracies due to the lack of experience and understanding of typosquatting by the crowd.

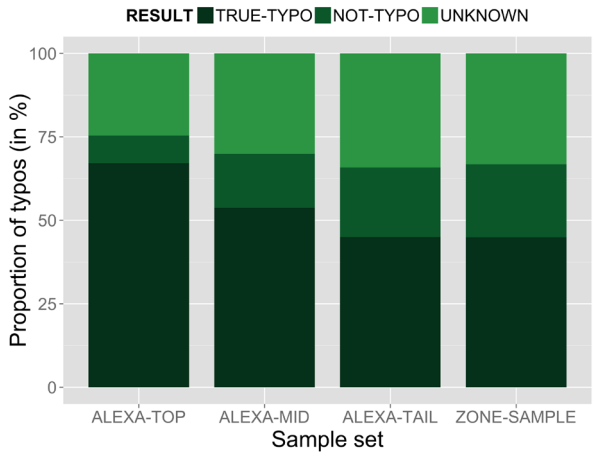


Figure 3: The prevalence of true typo domains in the four sample sets drawn popular and less popular `.com` domain names. The domain sets are ctypo samples of the Alexa top/mid/tail domains and the domains in the `.com` zone file. The number of true typo domains decreases with the Alexa rank of original domains, yet their ratio in the whole population remains high.

According to our manual inspection, a majority of the ctypo domains registered against the Alexa top domains are true typo domains (as shown in Figure 3). This result confirms the finding of [26]. We note here that there is a significant number of ctypo domains for which we cannot reliably decide if they are typo domains or not (U). This is mostly due to the fact that domains return "not accessible" responses for DNS or HTTP queries. The number of true typo domains steadily decreases when we perform the same experiment for the Alexa mid and tail domains, yet it remains high (around 50% within the set of all ctypo domains). While this indicates that thousands of domains are indeed typosquatting on less popular domains, to present defenses we need to develop a more reliable strategy to predict whether a domain is involved in typosquatting.

4.2 Accuracy of identification

We developed an automatic categorization tool based on the domain features presented in Section 3.5 called *Yet Another Typosquatting Tool (YATT)*. YATT has three modes. In the *passive mode, YATT-P* uses the information readily available from static files, such as lexical features, zone information and Alexa information. In the *DNS mode, YATT-PD* includes Whois and DNS features collected from the active crawler infrastructure, and finally in the *content mode, YATT-PDC* content features obtained via crawling are added to the categorization. The complexity of the algorithms increases from YATT-P to YATT-PDC. We expect that YATT-PDC will show the best performance in categorizing typo domains, but the other variants can still provide useful information if one wants to avoid the tedious work of collecting content features.

As presented before, we fine-tuned the parameters of YATT, but further improvement might be possible with additional features and a more complex feature selection process. At the moment, this optimization is left as future work.

In addition to YATT, we tested notable typosquatting identification methods from related work. First, we consider the method in [26], which showed that most ctypo domains of DL-1 are indeed true typos. Their primary feature is the domain length so we repeat their experiment for DL-1 and we name their method *AllTypo*. Then, we implemented the most important features of the *SUT-net* algorithm in [10] and compared it to various modes of YATT.

In Figure 4, we compare the accuracy of the typo identification methods in related work and the three modes of YATT to the established benchmark of manual evaluation. We perform this accuracy evaluation on the four ctypo domain samples described in Section 4.1. In Figure 4, we see that all five algorithms mark ctypo domains

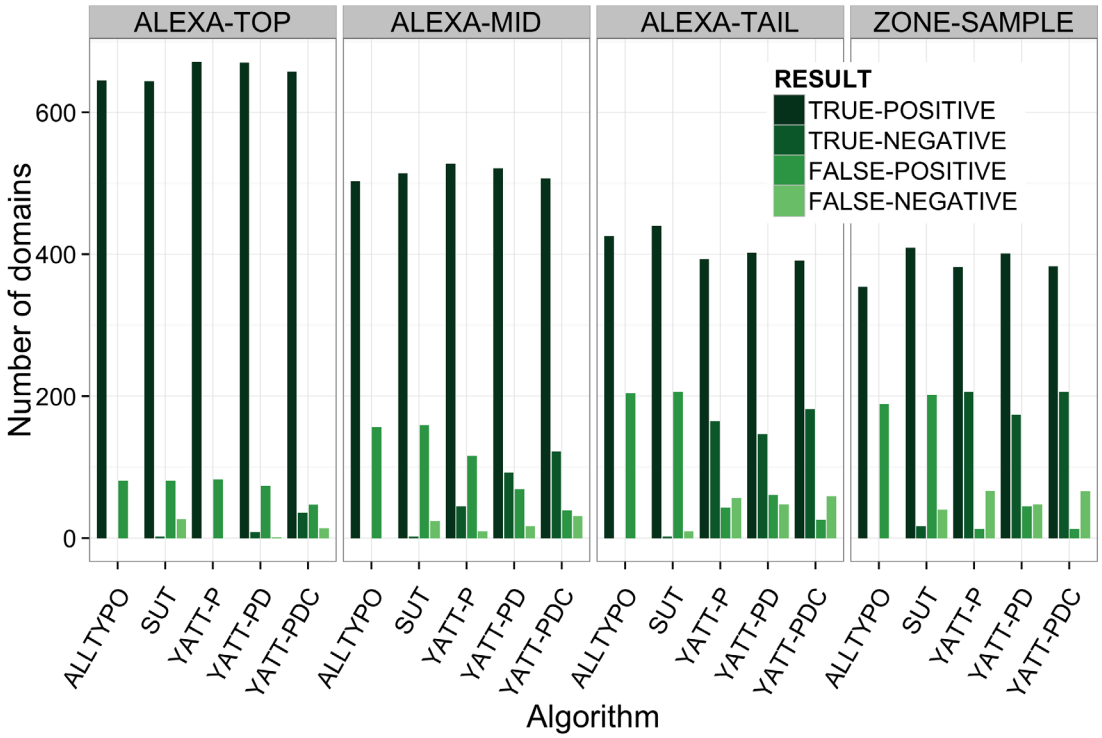


Figure 4: Accuracy of four typosquatting prediction tools. We tested (a) AllTypos, (b) SUT-net-based content features, (c) YATT-P, (d) YATT-PD, and (e) YATT-PDC for the four ctypo domain sample sets of (1/2/3) the Alexa top/mid/tail domains and (4) the domains in the `.com` zone file.

as positives in the Alexa top dataset. This assertive categorization results in a good true positive (TP) rate, a reasonably small number of false positives (FP) and with almost no false negatives (FN). Only the full YATT-PDC can identify a small set of true negatives (TN) in the population. In the Alexa mid, the agressive typo identification of AllTypo and SUT results in a high FP number whereas YATT keeps the FPs low while correctly identifying TNs (with YATT-PDC being the most accurate as expected). For the Alexa tail and zone datasets, the number of true typos further decreases and both AllTypo and SUT overwhelmingly categorize these domains as typos resulting in a very large false positive rate. All versions of YATT keep the FPs low and correctly categorize TNs at the expense of a small number of FNs. It is clear that perfect categorization is difficult to do, but YATT does not sacrifice much precision as the number of non-typo domains get introduced.

Next, we study the accuracy of the YATT-PDC to identify parked domains and other typosquatting indicators based on our manual sampling in Table 1. Note that related work on typosquatting identification usually focuses on typo identification and leaves the categorization aside. Only the active mode of the algorithm can perform this categorization, because it requires content features. YATT-PDC uses regular expression-based matching for the identification of parking domains. It matches these domains with about 85% precision, the error stemming from the incompleteness of the set of regular expressions we use. YATT-PDC still finds the majority of the parking sites and lists a significantly larger number of parking sites than methods in related work [7, 19]. For the defenisve domain registrations, YATT-PDC fares worse. It only finds 60-85% of the defensive registrations. This is due to the complexity of defensive registration patterns that can mostly be caught by a human eye. Finally, for affiliate registrations, YATT-PDC performs quite well, correctly categorizing almost all domains. We also checked the existence of malicious and phishing domains in our sample dataset, but we could not find any in such a small sample. Our results from more rigorously checking for maliciousness in typo domains is described in subsection 4.6, however maliciousness was not used to classify typo domains as typos.

YATT results in an accurate prediction of true typo domains and domain categories for the whole range of the domain population and hence its results can be used as a basis for intervention attempts and tools. Using YATT, we compile a typosquatting blacklist and use it in a set of mitigation tools (see Section 5).

	PARKED			DEFENSIVE			AFFILIATE		
	False Positive	True Positive	False Negative	False Positive	True Positive	False Negative	False Positive	True Positive	False Negative
Alexa top	3	402	76	0	39	15	0	27	1
Alexa mid	3	358	50	0	18	3	0	15	0
Alexa tail	1	295	59	0	9	3	0	0	0
Zone	0	265	43	1	7	4	0	0	0

Table 1: The accuracy of YATT to identify parked, defensive and affiliate registrations across the sample datasets.

4.3 Presence of typosquatting registrations

Having designed an accurate typosquatting identification tool, we now study the existence of typosquatting in current domains registrations. We first obtained 4.7 million ctypos targeting the `.com` domains in the Alexa top 1m domain list and existing in the `.com` zone file using the methodology described in Section 3. Recall, that we split the original domains according to their Alexa rank into the Alexa top/mid/tail categories.

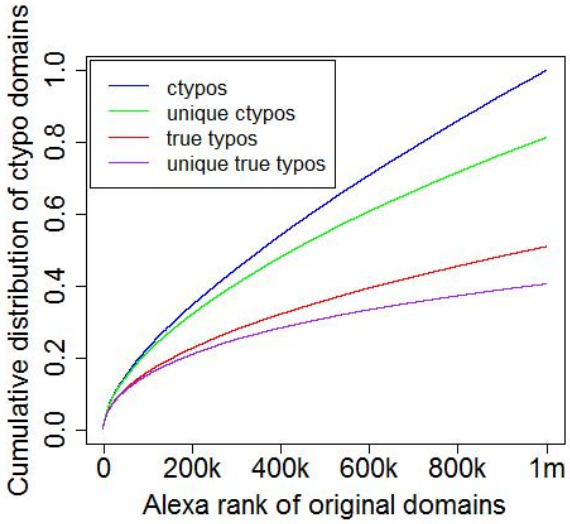


Figure 5: The cumulative distribution of true typo domains in ctypos and unique ctypos as a function of the Alexa rank of the original domains.

The first and foremost question is the extent of typosquatting targeting the Alexa domain set. We use YATT to determine typosquatting behavior and partition ctypo domains into the categories described in Section 3.5. In Figure 5, we plot the cumulative distribution of ctypo domains as a function of the originals' Alexa rank, and we also plot the cumulative distribution of true typo domains. We see that the number of true typos steadily increases as the Alexa rank decreases, although at a slower pace than the number of ctypos. In addition, we also plot the cumulative distribution of unique ctypos and true typo domains.

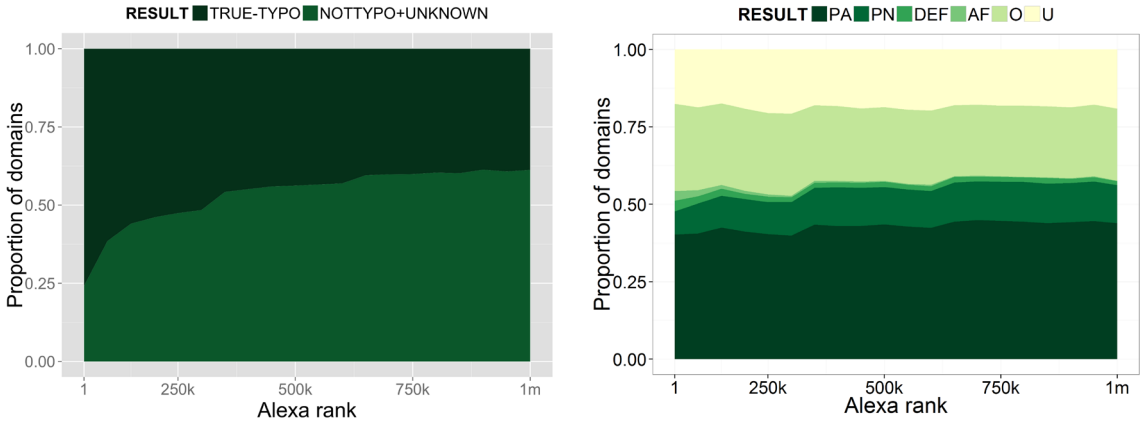
We then show the fraction of true typos in the population of ctypos in Figure 6(a). We calibrated YATT to make a decision about each ctypo and thus it conservatively categorizes the majority of unknown domains as not typos. For Alexa top sites, the fraction of true typos is higher, but for lower Alexa ranks the number of not-typo and unknown domains increases. This is consistent with our benchmarking results in Figure 3. Finally, in Figure 6(b), we present the typosquatting categories as a function of the original domains' Alexa rank. We observe that the bulk of the true typo registrations profits from parked domains with advertisements. The number of defensive and affiliate registrations is higher for the Alexa top sites, but then then the affiliate registrations disappear as we head to the Alexa tail while the defensive registrations persist. Finally, there is a significant number of non-typo domains incidentally close to the domains in the Alexa domain list.

Projecting our results to the total number of `.com` domains in the zone file, we estimate that about 53% of them are candidate typo domains and hence 20% of the total domain set are true typo domains. Based on our results, we estimate that about 21.2m domains are true typo domains in the `.com` zone file.

4.4 Trend analysis

We analyzed trends in typo domain registrations for a period of approximately one year (from 2012-10-01 to 2013-10-15). We considered domains from four datasets: domains from the `.com` zone file, ctypos from the `.com` zone file, ctypos targeting the whole Alexa list and ctypos targeting the Alexa top list.

For the purposes of our analysis, we use visibility into the `.com` zone file as a proxy for domain registration. Because the actual registration and registration lapse events are not visible to us, we use presence in the zone file as a proxy for registration events. We define a registration



(a) (b)

Figure 6: The existence of typosquatting domains targeting the Alexa domain set. The fraction of (a) true typo domains and (b) various typo categories in the true typo population.

event as one where a domain was not in a daily zone dump, and was present in the subsequent day's zone file, and vice-versa for a registration lapse, or deregistration.

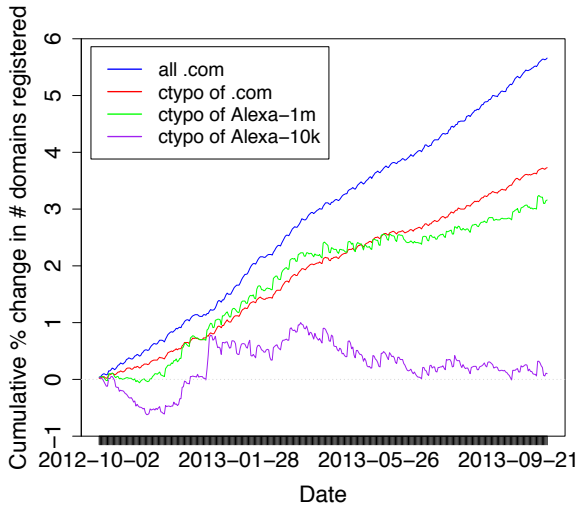


Figure 7: Cumulative change in the total number of domains registered over time.

We looked at the change in domain registrations over time. Figure 7 plots the cumulative changes in the number of domains registered in the above mentioned domain sets. While the overall registration rate is steady, the difference between the rate of Alexa-10k targeted and Alexa-1m targeted typos suggests that, through enforcement or typosquatter preference, the overall increase in registrations targeting popular domain typos is far smaller even though

	Stable	Mean uptime	Reregs
Alexa-1m ctypo	72.3%	458 days	49.5%
Alexa-10k ctypo	71.0%	454 days	49.5%
Alexa-1m	93.3%	501 days	67.1%
Alexa-10k	99.0%	506 days	86.8%
Random sample	70.4%	440 days	28.5%

Table 2: Speculation trend analysis between 2012-10-01 and 2014-02-20. Alexa list and zone file used was from 2012-10-01. The "stable" column indicates what proportion were registered throughout the dataset. "Reregs" indicates how many domains experienced at least one lapse in visibility at the zone file, indicating that the domain was decommissioned and then reactivated. "Random sample" is a selection of 2 million random domain names from the `.com` zone file of 2012-10-01.

many DL-1 typos of popular domains are still available. It is also interesting to note that the spike centered on January 1 2013 is due to four organizations (sedoparking, 1and1.com, dsredirection, and graceperioddomain.com) registering a large number of domains: these four account for 87% of all domains registered at that time.

Our next analysis focuses on the amount of speculation present within the market for typosquatting domains between 2012-10-01 and 2014-02-20. Table 2 shows the percentage of stable domains, the average uptime, and the percent of domains experiencing at least one reregistration event during our measurement time period. As might be expected, random domains are purchased and left to lapse very often, with less than one third being reregistered after being abandoned. Domains which are a typo of a popular domain, however, experience almost

twice as much interest, although they are not active for significantly more time. This trend suggests that the information asymmetry of the typosquatting marketplace is such that new speculators register old typos at a much higher rate than random domains.

4.5 Typosquatting redirections

Now, we scrutinize the affiliate redirections via third-parties. This third-parties can be legitimate brand protection companies, but more frequently they are typosquatting affiliates collecting type-in traffic from a large number of typo domains.

Domain redirections that lead back to the targeted original domains without intermediate domains are considered defensive registrations, as explained in Section 2.1. If the redirection leads back to the target domain via a third-party, then we call it an affiliate defensive registration. In Figure 8 the first graph shows that in the cumulative distribution of third party landing pages, eleven domains (less then 0.1 percent of all of these landing pages) get redirections from more than 50 percent of ctypos redirecting to a third party domain. The second graph in Figure 8 shows defensive affiliate domains, where the landing pages is the original domain, but the traffic goes through an intermediate affiliate domain. 18 such intermediate domains (1.3 percent of all domains) are responsible for more than 80 percent of defensive affiliate marketing. Even though this set has a very small overlap with the non-defensive affiliate domains, a small fraction of affiliate domains are controlling 80 percent of the affiliate market.

Finally, if the redirection leads to a third-party domain, that is away from the original target, then this is considered truly speculative. The third graph in Figure 8 shows redirections to third-party pages with only one redirection. Here the domains are more widely distributed: there is only one big landing domain hugedomains.com which receives traffic from more than 21 percent of this type of redirection. The last graph shows the cumulative distribution of all affiliate domains participating in third-party redirections with a non-defensive purpose. That means that these affiliate domains lead away the users from the targeted original sites. 41 of these non-defensive affiliate domains (0.4 percent of all such domains) control the traffic originating from more than 80 percent of candidate typo domains. This means that, here too, a relatively small set of domains control the majority of such traffic going to a few landing pages.

4.6 Maliciousness of Typo Domains

In order to test the assertion that typo domains are more malicious than other domains, the candidate typo (ctypo) and true typo (ttypo) domains extracted from the `.com` were checked against a variety of available black lists. These results are compared against the same test on the Alexa domains. By using 12 available black lists from various sources fluctuations due to the idiosyncrasies of any individual list can be controlled.

	# Malware Hits	% of List Marked Malware	# Phish Hits	% of List Marked Phish
Alexa	9990	1.907%	27	0.005153%
ctypos	17485	0.3716%	272	0.005781%
ttypos	3720	0.1585%	125	0.005329%

Table 3: Google Safe Browsing results for domains in Alexa, *ttypos*, and *ctypos*.

The Alexa top 488,133 `.com` domains (all the `.com` domains in the top 1m) are more likely to appear on black lists than the typos of them, either *ctypos* or *ttypos*. This result is consistent across all 12 black lists investigated. In each case, the Alexa domains are more likely to host malicious activity. The percentage of .com domains from the Alexa list on each black list is always higher than the percentage of *ttypo* domains on the same list.

Google's Safe Browsing list requires a different checking method, due to their storage method. The list also distinguishes between a match due to malicious content or attempts at phishing. However, the results show a similar trend. The Alexa domains are more likely to be purveyors of malicious software. Table 3 shows the results for Google Safe Browsing checking for any listing from May 1, 2011 – July 31 2013.

There are several possible causes for this pattern, and several of them would be uninteresting. A possibility is that there is a pocket of malicious activity using typos, but that most of it is benign. The first place to look for this would be the name servers hosting predominantly typo domains. There are 10 name servers for which most of the domains they host are typos of other domains—for these name servers, between 20-80% of their domains are typos.

The typo domains hosted on these 10 name servers seem to be even less likely to appear on a black list. The average percentage of these name servers' domains on any of the black lists is 0.051%, and the maximum percentage of typo domains hosted by one of these name servers on any one list is 0.27%. Both of these numbers are below those both for typos generally as well as the results for the Alexa domains.

5 Intervention options

Just as defining typosquatting remains one of the grey areas of domain name security, developing effective in-

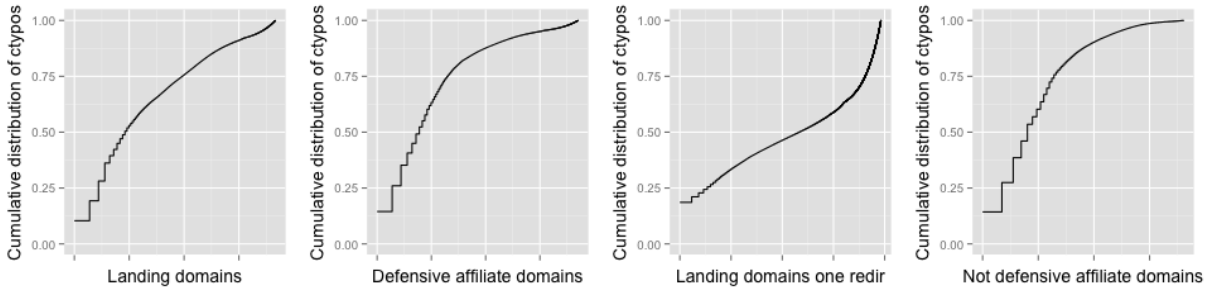


Figure 8: The leftmost figure shows the cumulative distribution of landing pages targeted from ctypo domains. The second figure shows the cumulative distribution of intermediate domains in case of defensive redirections. The third figure is when the length of the domain redirection chain is one. Finally, the rightmost figure shows the cumulative distribution of intermediate domains in case of redirections targeting a third party.

tervention techniques is similarly difficult. So far, most intervention attempts remain ineffective. In the following, we present viable typosquatting mitigation options and present a set of practical tools to prevent typosquatting from negatively effecting users.

5.1 Policy intervention

Much of the effort to crack down on typosquatting focuses on policy options. Two major tools exist for policy intervention. The first is the UDRP arbitration framework provided by ICANN [21]. Unfortunately, only a small fraction of typosquatting domains enters the UDRP procedure [26], although domains are claimed by their trademark holders very often.

The Anti-cybersquatting Consumer Protection Act (ACPA) (15 USC §1125(d)) offers an alternative to the UDRP through legal action. The act "was designed to thwart cybersquatters who register Internet domain names containing trademarks with no intention of creating a legitimate web site, but instead plan to sell the domain name to the trademark owner or a third party." While originally aimed at preventing cybersquatting, in May 2013 Facebook successfully litigated a case including typosquatting domains, earning a US $2.8 million judgement [18]. As with any legal action, the enforcement of this act is costly and only major trademark holders have exercised their legal rights [25, 31, 32]. Additionally, the bad faith of typosquatting registrations is difficult to prove and hence the legal action might not always be efficient [30]. Unfortunately, even vigilant companies seem overwhelmed by the number of typosquatting domains targeting their brands, motivating them to litigate; even so, many of their domains are still controlled by typosquatters.

5.2 Infrastructure support

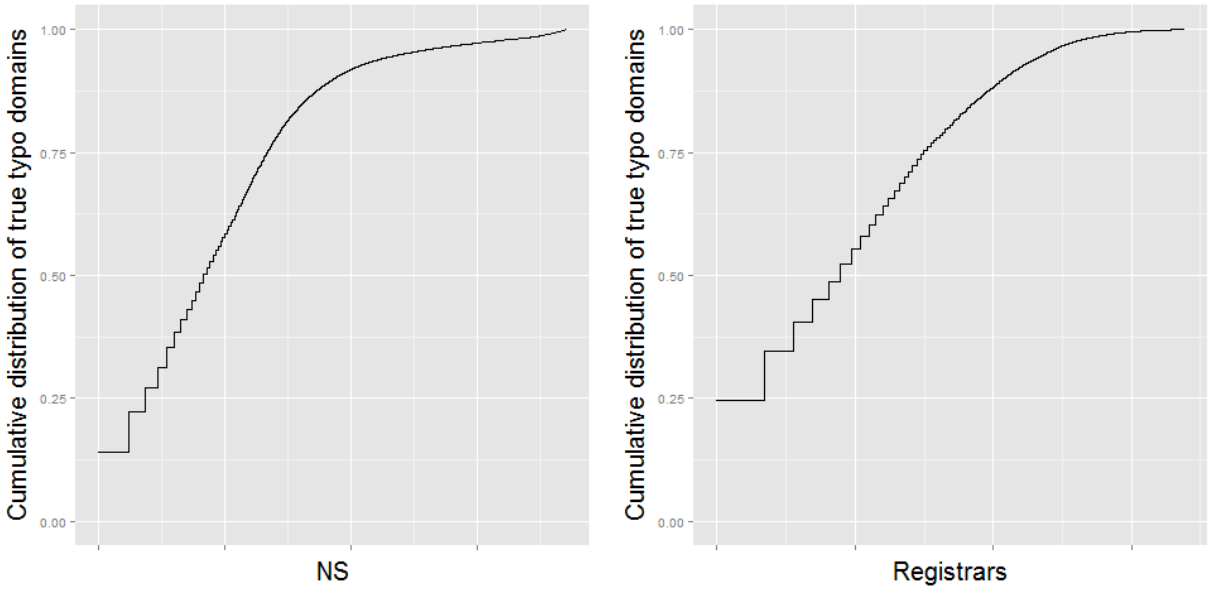
Another option for intervention is to motivate registrars and hosting providers to scrutinize domain name registrations when they happen (with a mandatory light-weight UDRP procedure for example). Let us now look at the potential of registration intervention at the infrastructure side. Figure 9 shows the distribution of typosquatting domains (a) as a function of the registrars and (b) as a function of the supporting NSs (setting the x axis to a log scale to improve visibility). We observe that most true typo domains cluster at major registrars and are hosted at a few NSs. In particular, 12 NSs and 5 major registrars are responsible for hosting 50% of the true typo domains. Forcing these major registrars to enforce prudent registration practices with respect to typosquatting may be a viable policy option.

NS	True typos	All domains	Typo ratio
`a0f.net`	5221	6332	82%
`citizenhawk.net`	8819	12004	73%
`easily.net`	18281	36890	50%
`domainingdepot.com`	51854	132864	39%
`next.org`	9426	30252	31%
`domainmanager.com`	23493	90929	26%

Table 4: Worst offender NSs in true typo hosting with at least 5000 true typo domains. All NSs in the top list have higher than 25% of true typo / all domain ratio.

Based on the `.com` zone file, we are also able to collect the ratio of true typo domains to the total number of domains. Table 4 presents the top offenders with at least 5000 true typo domains hosted. Interestingly, there are only 65 NSs with such a high number of true typo domains. We see that the worst offenders almost exclusively host true typo domains, and none of them belong to the major hosting companies[8]. Further investigating

[8] An interesting case might be `citizenhawk.net`, a brand protec-



(a) (b)

Figure 9: Intervention potential at domain registrars and hosting companies. We present the distribution of typosquatting domains (a) as a funtion of the registrars and (b) as a function of the supporting NSs (while setting the x axis to a log scale for better visibility)

these typo domains we found two interesting results. First, out of the 6 name servers with the highest true typo ratio, 5 have domains that are privately registered and only `citizenhawk.net` is not, showing that the others are aware that their monetization strategy is questionable. Second, we found that on the average 24.5 percent of the domains hosted by these NSs is in the top Alexa, which is 2.5 time higher number than for the rest of the name servers. This indicates that these name servers are more effectively targeting popular typo domains than major hosting services who are not focusing on typosquatting. These hosting companies with an unusually high number of true typo domains could be regulated to effectively decrease the effect of speculative typosquatting.

Infrastructure intervention is promising if it can be enforced globally by ICANN on the supporting providers. Unfortunately, it is unlikely that such a global action will emerge as this is counterproductive for the domain registrars, and thus miscreants can always shift their businesses to negligent or accomplice providers who are financially motivated to assist their businesses. Registrar- and hosting-level intervention remains ineffective against spammers [23, 24] and it is unlikely that it will be effective against typosquatting. Registrars and hosting companies do not suffer from typosquatting, thus there is little economic incentive for them expend resources to defend against it.

tion company who probably registered a large number of domain names for protecting their customers.

5.3 Mitigation tools

The last option to counter typosquatting is the application of technical tools to reduce the impact of typosquatting. There exist mitigation tools to this end, but most tools suffer from either trivial errors or from small coverage of typosquatting domains.

Related work. Wang et al. developed *Strider Typopatrol*, a tool to automatically discover typo domains of popular domains [35]. They focus on a small subset of the Alexa top domain list [1], phishing targets, and childrens' websites. OpenDNS [27] provides typosquatting correction in their DNS services, but only for major TLDs. A similar tool called *URLFixer* [6] was introduced in the Adblock Plus advertisement blocking tool. The URLFixer tool includes misspellings of top Alexa domains, but fails to correct less popular domain names and includes some short domain names leading to false corrections. Chen et al. [11] develop a browser plugin to check typo domains based on a user-customized local repository. Banerjee et al. [9, 10] propose *SUT*, a method to identify typosquatting domains mostly based on HTML properties. Finally, the autocomplete feature of most major browsers can also decrease the instance of typos, albeit only for previously visited sites.

Initial tests show that most existing solutions are lim-

ited in scope (the most popular domains or most frequent typos), in features (only TLD correction or HTML features) or in the information used (search typing or local browser history) and consequently these tools are missing a large set of typosquatting domains.

The YATT framework. We developed a typosquatting categorization tool, YATT, that uses an extended domain feature set to provide accurate typosquatting identification. Based on the output provided by YATT, we implemented three typosquatting detection and protection services. The first service is a typosquatting blacklist (YATT-BL) compiled from the output of one of the versions of the YATT tool. As a DNS based blacklist, this access method is quick and lightweight. The tool works similarly to major domain blacklists such as URIBL [5], SURBL [4] or the Spamhaus DBL [3] and it can be used to filter out typo domains from live traffic. The DNS server uses RPZ [34] to efficiently distribute the typo list.

Second, we implemented a Firefox browser plugin and a corresponding typo protection server to protect users from typosquatting domains. Our plugin contacts the typo protection server each time a user types in a domain and raises a warning if the domain typed by the user is found on the typosquatting domain list. The user is provided with the option of accepting the automatic correction or rewriting it to her needs. The typo protection server uses YATT-BL DNS blacklist described above.

Third, we are in the process of implementing a YATT DNS server for organizations that want to avoid typosquatting yet do not want to expose their DNS traffic to a third party server. Using this tool, a company could periodically download an updated typosquatting blacklist and query it locally.

6 Conclusion

Typosquatting has caused annoyances for Internet users for a long time. Since users lack effective countermeasures, speculators keep registering domain names to target domains and exploit the traffic arriving from mistyping those domain names. Existing studies of typosquatting focused on popular domain names and thus have only shown the tip of the iceberg. Similar to traditional cybercrimes like spamming or financial credential fraud, typosquatting has minimal transparency, allowing what may be an unprofitable activity to continue because new entrants see its effects and attempt to become profitable typosquatters themselves. Investigating such speculative, "gray area" behavior longitudinally can give us insights which might generalize to traditional cybercrime and cybercriminals.

In this paper, we performed a thorough study for an extensive set of potential target domains. We found that 95% of typo domains are targeting less popular domains. We designed an accurate typo categorization framework and find that typosquatting using parked ads and similar monetization techniques not only exists for popular domains, but a whole range of domain names in the Alexa domain list. We showed that a large number of incidental domain registrations exist with close lexical distance to the target domains. Our conservative estimates indicate that as much as 21.2 million `.com` domain registrations are confirmed true typo domains, which accounts for about 20% of all `.com` domain registrations. Additionally, we found that the typosquatting phenomenon is only continuing to thrive and expand.

The difficulty of categorizing typosquatting domains partially explains the inefficiency of existing mitigation techniques. Much like typosquatting itself, mitigation is a gray area: one cannot easily classify a new registration as an example of typosquatting based on the name alone. As such, typo domains rarely appear on blacklists. To counter this problem, we designed several defense tools that rely on a broad range of features. We provide a typosquatting blacklist and a corresponding browser plugin to prevent mistyping at the user side. While typosquatting will likely continue to exist, these analyses and tools may improve user experience and further decrease the profit available to typosquatters.

7 Acknowledgements

We thank the anonymous reviewers and the PC at large for their helpful feedback. We also thank Nicolas Christin for his support. This work was made possible in part by the National Science Foundation grant NSF CNS-1351058. Mark Felegyhazi was supported by the Bolyai Janos Research Fellowship Nr: BO/00273/12. Janos Szurdi has been supported by the Ann and Martin McGuinn Graduate Fellowship.

References

[1] Alexa top sites. http://www.alexa.com/topsites.

[2] PhishTank. http://www.phishtank.com.

[3] Spamhaus DBL. http://www.spamhaus.org/dbl/.

[4] Surbl domain blacklist. http://www.surbl.org/lists.

[5] Uribl domain blacklist. http://www.uribl.com/about.shtml.

[6] Urlfixer for mozilla firefox. http://urlfixer.org/.

[7] ALMISHARI, M., AND YANG, X. Ads-portal domains: Identification and measurements. *ACM Transactions on the Web (TWEB) 4*, 2 (2010), 4.

[8] AVAST. Misspelling goes criminal with typosquatting. https://blog.avast.com/2012/03/23/misspelling-goes-criminal-with-typosquatting/, Mar 23 2012.

[9] BANERJEE, A., BARMAN, D., FALOUTSOS, M., AND BHUYAN, L. N. Cyber-fraud is one typo away. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE* (2008), IEEE, pp. 1939–1947.

[10] BANERJEE, A., RAHMAN, M. S., AND FALOUTSOS, M. SUT: Quantifying and mitigating url typosquatting. *Computer Networks 55*, 13 (2011), 3001–3014.

[11] CHEN, G., JOHNSON, M. F., MARUPALLY, P. R., SINGIREDDY, N. K., YIN, X., AND PARUCHURI, V. Combating typo-squatting for safer browsing. In *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on* (2009), IEEE, pp. 31–36.

[12] COULL, S. E., WHITE, A. M., YEN, T.-F., MONROSE, F., AND REITER, M. K. Understanding domain registration abuses. *Computers & security* (2012).

[13] DAMERAU, F. J. A technique for computer detection and correction of spelling errors. *Communications of the ACM 7*, 3 (1964), 171–176.

[14] DANCHEV, D. Legitimate software typosquatted in SMS micro-payment scam. blog, http://ddanchev.blogspot.com/2009/07/legitimate-software-typosquatted-in-sms.html, Jul 7 2009.

[15] EDELMAN, B. Large-scale registration of domains with typographical errors. http://cyber.law.harvard.edu/people/edelman/typo-domains/, Sep 2003.

[16] EDELMAN, B. Estimating visitors and advertising costs of typo domains. http://www.benedelman.org/typosquatting/pop.html, 2010.

[17] FELEGYHAZI, M., KREIBICH, C., AND PAXSON, V. On the potential of proactive domain blacklisting. In *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more* (2010), USENIX Association, pp. 6–6.

[18] GROVE, J. V. Facebook wins millions in case against typo squatters. http://www.cnet.com/news/facebook-wins-millions-in-case-against-typo-squatters/, 2013.

[19] HALVORSON, T., SZURDI, J., MAIER, G., FELEGYHAZI, M., KREIBICH, C., WEAVER, N., LEVCHENKO, K., AND PAXSON, V. The BIZ top-level domain: ten years later. In *Passive and Active Measurement* (2012), Springer, pp. 221–230.

[20] HIDAYAT, A. Phantomjs. http://phantomjs.org/, 2013.

[21] ICANN. Uniform domain name dispute resolution policy (UDRP). http://www.icann.org/en/help/dndr/udrp, 1999.

[22] ICANN. The end of domain tasting - status report on AGP measures. http://www.icann.org/en/resources/registries/agp/agp-status-report-12aug09-en.htm, Aug 12 2009.

[23] LEVCHENKO, K., PITSILLIDIS, A., CHACHRA, N., ENRIGHT, B., FELEGYHAZI, M., GRIER, C., HALVORSON, T., KANICH, C., KREIBICH, C., LIU, H., ET AL. Click trajectories: End-to-end analysis of the spam value chain. In *IEEE Symposium on Security and Privacy, 2011* (2011), IEEE, pp. 431–446.

[24] LIU, H., LEVCHENKO, K., FELEGYHAZI, M., KREIBICH, C., MAIER, G., VOELKER, G. M., AND SAVAGE, S. On the effects of registrar-level intervention. In *Proceedings of 4th USENIX LEET* (2011).

[25] MICROSOFT TECHNET. The trouble with typosquatting. http://blogs.technet.com/b/microsoft_on_the_issues/archive/2010/04/15/the-trouble-with-typosquatting.aspx, Apr 15 2010.

[26] MOORE, T., AND EDELMAN, B. Measuring the perpetrators and funders of typosquatting. In *Financial Cryptography and Data Security* (2010), Springer, pp. 175–191.

[27] OPENDNS. There's no "i" in twtter: How to outsmart typosquatting. http://blog.opendns.com/2011/09/02/there%E2%80%99s-no-%E2%80%9Ci%E2%80%9D-in-twtter-how-to-outsmart-typosquatting/, Sep 2 2011.

[28] SOGHOIAN, C., FRIEDRICHS, O., AND JAKOBSSON, M. The threat of political phishing. In *The International Symposium on Human Aspects of Information Security & Assurance* (2008), Citeseer.

[29] SOPHOS, NAKED SECURITY. Typosquatting - what happens when you mistype a website name? http://nakedsecurity.sophos.com/typosquatting/, Dec 14 2011.

[30] SUNDERLAND, S. D. Domain name speculation: Are we playing whac-a-mole. *Berkeley Tech. LJ 25* (2010), 465.

[31] TECHCRUNCH.COM. U.S. Court Rules For Facebook In Its Case Against Typosquatters On 105 Domains; $2.8M In Damages. http://techcrunch.com/2013/05/01/u-s-court-rules-for-facebook-in-its-case-against-typosquatters-on-105-domains-2-8m-in-damages/, May 1 2013.

[32] THE NEXT WEB. Typosquatting sites 'wikapedia' and 'twtter' have been fined $300,000 by UK watchdog. http://thenextweb.com/insider/2012/02/16/typosquatting-sites-wikapedia-and-twtter-have-been-fined-300000-by-uk-watchdog/, Feb 16 2012.

[33] THE REGISTER. Typosquatters set up booby-trapped High Street names. http://www.channelregister.co.uk/2011/12/13/typosquatting_scams_target_xmas_shoppers/, Dec 13 2011.

[34] VIXIE, P. Taking back the DNS. http://www.isc.org/community/blog/201007/taking-back-dns-0, Jul 29 2010.

[35] WANG, Y.-M., BECK, D., WANG, J., VERBOWSKI, C., AND DANIELS, B. Strider typo-patrol: discovery and analysis of systematic typo-squatting. In *Proc. 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)* (2006).

[36] WEAVER, N., KREIBICH, C., AND PAXSON, V. Redirecting DNS for ads and profit. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI), San Francisco, CA, USA (August 2011)* (2011).

[37] WEBSENSE SECURITY LABS. The rise of a typosquatting army. http://community.websense.com/blogs/securitylabs/archive/2012/01/22/The-rise-of-a-typosquatting-army.aspx, Jan 22 2012.

Appendices

A Features used for domain categorization

Feature description	Priority	Comment
Lexical attributes		
domain length	M	[26]
highest-ranked neighbor's operation	M	diff. from the most popular original domain
is any neighbor at fat finger distance one?	M	FF typos are more likely to be true typos [26]
nr. of neighbors	L	
nr. of neighbors with *op*	L	where *op*={*add*,*del*,*sub*,*tra*,*www*}
Popularity (Alexa) attribute		
Alexa rank of original domain	H	
Zone file attributes		
total nr of ctypo-s on NS	M	
ctypo/alldomain ratio on NS	H	
total nr. of domains on the NS in the zone	L	
parked keywords in NS domain	H	
Whois attributes		
total nr of ctypo-s at registrar	M	
registration date	L	
DNS attributes		
NXDOMAIN wildcarding	H	
TXT google auth	L	Google ads affiliate auth
total nr of ctypo-s on IP address	M	[10]
Content attributes		
Parked	H	by RE keywords
Serving ads	M	by RE keywords
Total redirection length	M	# of redirections [10]
Domain redirection length	H	# of redirections between registered domains
DERPContent size	M	[10]
Affiliate marketing	M	[26]

Table 5: Domain and infrastructures features to categorize candidate typo domains. The column Priority indicates the relative importance in idenfitying typosquatting behavior.

Understanding the Dark Side of Domain Parking

Sumayah Alrwais[1,2], Kan Yuan[1], Eihal Alowaisheq[1,2], Zhou Li[1,3] and XiaoFeng Wang[1]

[1]Indiana University, Bloomington
{salrwais, kanyuan, ealowais, xw7}@indiana.edu
[2]King Saud University, Riyadh, Saudi Arabia
[3]RSA Laboratories
zhou.li@rsa.com

Abstract

Domain parking is a booming business with millions of dollars in revenues. However, it is also among the least regulated: parked domains have been routinely found to connect to illicit online activities even though the roles they play there have never been clarified. In this paper, we report the first systematic study on this "dark side" of domain parking based upon a novel infiltration analysis on domains hosted by major parking services. The idea here is to control the traffic sources (crawlers) of the domain parking ecosystem, some of its start nodes (parked domains) and its end nodes (advertisers and traffic buyers) and then "connect the dots", delivering our own traffic to our end nodes across our own start nodes with other monetization entities (parking services, ad networks, etc) in-between. This provided us a unique observation of the whole monetization process and over one thousand *seed* redirection chains where some ends were under our control. From those chains, we were able to confirm the presence of click fraud, traffic spam and traffic stealing. To further understand the scope and magnitude of this threat, we extracted a set of salient features from those seed chains and utilized them to detect illicit activities on 24 million monetization chains we collected from leading parking services over 5.5 months. This study reveals the pervasiveness of those illicit monetization activities, parties responsible for them and the revenues they generate which approaches 40% of the total revenue for some parking services. Our findings point to an urgent need for a better regulation of domain parking.

1 Introduction

Consider that you are a domain owner, holding a few domain names that you do not have a better use of. Then one thing you could do is to "park" them with a domain parking service to earn some extra cash: whenever web users type in those domain names (probably accidentally) in the browser's address bar, the parking service resolves the domains to advertisement laden pages, the revenue generated in this way is then split between the parking service and the domain owner. Such *domain parking monetization* is a million-dollar business [29], offering a unique marketing channel through newly acquired, underdeveloped domains, or those reserved for future use. However, with a large number of parked domains being monetized through those services (`Sedo` reported to have 4.4M parked domains in 2013 [29]), what becomes less clear is the security implications of such activities, particularly whether they involve any illicit operations, a question we attempt to answer.

Dark side of domain parking. Prior research shows that once malicious domains (e.g., those hosting a Traffic Distribution System, TDS) have been discovered, they often end up being parked [17], i.e. temporarily hosted by some domain parking services. Those domains come with a large number of backlinks through which they can still be visited by the victims of the malicious activities they were once involved in, such as compromised websites. Web traffic from those backlinks is clearly of low quality but apparently still used by the parking services to make money [17]. More problematically, a recent study reveals suspicious redirections performed by some parking services could be related to click spam [10], though this could not be confirmed. This finding echoes what we observed from the redirection chains collected by crawling parked domains, some of the URLs on the chains carried URL patterns related to ad click delivery even though our crawler did not click on any link at all (Section 2.3). Also malicious web content has long been known to propagate through parked domains [14, 20].

With all such suspicions raised, it is still challenging to determine whether some parking services are indeed involved in illicit activities, and if so the types of roles they play there. The problem is that parking services' monetization decisions and strategies cannot be directly observed from the outside. Therefore, in the absence of information about the nature of input traffic to those services and the actual ways it has been monetized, all we

have is guesswork. For example, even though we did observe redirection chains seemingly related to click delivery, without knowing a parking service's interactions with its ad network, we were still left with little evidence that what we saw was indeed click fraud. Note that parking services do not need to play conventional tricks, such as running click bots [21] or using hidden iframes embedded in a compromised page, to generate fraudulent clicks, and therefore will not be caught by standard ways in click fraud detection. In the presence of such technical challenges, little has been done so far to understand the illicit activities that can happen during the monetization of parked domains.

Our study. In this paper, we report the first attempt to explore the dark side of domain parking and uncover its security implications. This expedition is made possible by an innovative methodology to infiltrate parking services. More specifically, we purchased a set of domains and parked them with those services. Those domains, together with our crawler that continuously explored parked domains, enable us to control some inputs to the parking services. On the receiving end, we launched ad campaigns and made purchases of web traffic through the ad network or the traffic sellers associated with those parking services. By carefully tuning parameters to target web audience we controlled, we were able to connect the dots, receiving the traffic generated by our crawler going through our parked domains and onto our campaign websites. This placed us at a unique vantage point, where we could observe complete monetization chains between the start and the end nodes we controlled.

By analyzing such monetization chains, we were surprised to find that domain parking services, even highly popular ones such as `PS5`[1], shown in Table 1, are indeed involved in less-than-legitimate activities that should never happen: our ad campaigns were charged for the "click" traffic produced by our own crawler that never clicked and the traffic we purchased turned out to have nothing to do with the keywords we specified; also interestingly, we found that for all the visits through our parked domains and hitting our ads, only some of them were reported to our domain-owner's account (i.e. their revenues shared with us), though all of them were billed to our ad account.

To further analyze the scope and magnitude of those problems, which we call *click fraud*, *traffic spam* and *traffic stealing* respectively, we fingerprinted those confirmed illicit monetization chains with a set of salient features called *stamps*, and utilized them to identify monetization activities on 24M visits to parked domains not going through our end nodes (i.e. ad/traffic campaign websites). New findings reveal that even leading domain parking services are involved in illegitimate operations. On the other hand, such operations were present in only about 5% of the traffic we observed, which indicates that those services are largely legitimate. Possible motivations behind their opportunistic attacks could be monetizing less reputable (secondary) domains (e.g., taken-down malicious domains) that are difficult to profit from legitimately, or making up for revenue losses. Furthermore, we conducted an economic study to estimate the revenues of such dark-side activities, which we found to be significant.

Contributions. The contributions of the paper are outlined as follows:

• *New methodologies*. We performed the first systematic study of illicit activities in parked domain monetization. This study was made possible by a suite of new methodologies that allowed us to infiltrate domain parking services and collect a set of complete monetization chains. We further expanded such "seed" chains over a large number of redirection chains collected from parked domains over a 5.5-month period, which laid the foundation for understanding the unique features and the impacts of those activities.

• *New findings*. Our study brought to light a set of interesting and important findings never reported before. Not only did we confirm the presence of illegitimate operations including click fraud, traffic spam and traffic stealing during the monetization of parked domains, but we also reveal the pervasiveness of those activities which affect most leading parking service providers and attribute it to account for up to 40% of their total revenue. Also interesting is the discovery of unique features of those illicit activities and their relations with different monetization strategies and parking service syndicates.

2 Parked Domain Monetization

2.1 Background

Domain parking. As described before, a parked domain is a registered domain name whose owner does not have a better use of it than temporarily running it as an ad portal to profit from the traffic the domain receives. To this end, the owner typically chooses to *park* the domain with a domain parking service, an intermediary between the owner and various monetization options (explained later). This is done by setting up an account with the service, and the owner forwards her domain traffic to the parking service as specified by its regulations. The most common way for doing this is through the Domain Name System (DNS), in which the parked domain's Name Server (NS) or Canonical Name record (CNAME) is set to point to that of the parking service.

[1]Throughout this paper, we anonymize the identities of domain parking services found to be participating in illicit activities due to legal restrictions imposed by Indiana University and RSA.

In this way, the service gains complete control on the parked domain and any traffic it receives. Alternatively, the domain owner can choose to log her domain traffic before redirecting it to the parking service through HTTP redirections.

Parking services provide a domain owner with a platform to manage her parked domains. For example, some parking services let the owner set the keywords to be used for the parked domain monetization. Also, a domain owner can monitor their domain earnings through revenue reports.

Monetization options. A parked domain owner can profit from her domain traffic through a number of monetization options. The most popular ones are search advertising and direct-navigation traffic monetization, as elaborated below:

• *Search advertising*. In search advertising (aka., sponsored search), the advertiser runs a textual ad campaign with a search ad network and selects a set of target keywords for displaying her ads. To serve these ads, the publisher may operate search-related services such as search engines and toolbars, or use in-text advertising techniques (i.e. when the mouse hovers on a target word, the ad is displayed) to identify the right context for advertising: for example, it shows ads associated with the target words that are included in the search terms entered into search engines or the toolbars. Parking services are one of such publishers who provide textual ads relevant to the names of parked domains.

Search advertising is made possible through pay-per-click (PPC) XML feeds as illustrated in Figure 1. A publisher submits a search query for certain keywords (relevant to the domain names, in the case of parked domains) and receives relevant ads in the XML format, which also include the bidding price per ad from the advertisers. The publisher in turn picks up a set of ads to display. Once a user opts to click on an ad, the click traffic is bounced through a number of hosts such as click servers before reaching the advertiser's web page. This click is paid for by the advertiser and the revenue generated in this way is shared between the publisher and the ad networks.

Popular search ad networks such as `Google AdWords` and `BingAds` are considered to be top-tier (premium), while other less reputable ones are 2nd-tier or lower. Compared with other ad networks, top-tier networks offer a higher rate per click (CPC) to the publisher and a better click fraud detection to the advertiser.

• *Direct navigation traffic (PPR)*. Direct navigation traffic (aka., *type-in* traffic) is generated when the web user enters a domain name as a query and expects to be redirected to a related domain. For example, one may type in "findcheaphotels.com" in the address bar and land at `mytravelguide.com`. This is caused by

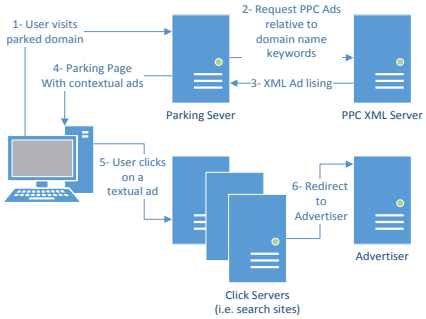


Figure 1: Legitimate PPC XML feed operation.

a direct-navigation-traffic purchase in which the owner of `mytravelguide.com` purchases through a direct navigation system the traffic related to keywords "travel" and/or "hotels", for example. Parked domains can serve such a direct navigation system by redirecting type-in traffic to them who ultimately redirect it to traffic buyers like `mytravelguide.com`. This monetization option is called Pay-Per-Redirect (PPR) or zeroclick.

2.2 Ecosystem and Illicit Monetization

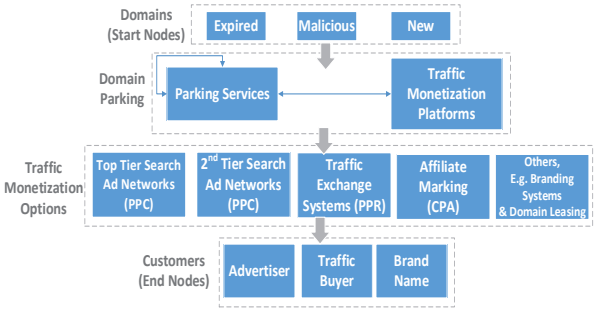


Figure 2: Domain Parking Ecosystem

Domain parking services have a significant hold in the domain industry. Table 1 shows some of the most popular parking services investigated in our study. By looking at the ranks of the domain names associated with the companies running these parking services, we find that many of them are ranked among the top 100k by Alexa [2]. Also shown in the table is the total number of parked domains observed on Feb. 25^{th} according to *DailyChanges* [34]. Note that the list maintained by DailyChanges is not comprehensive. Nevertheless, it is quite clear from Table 1 that parking services indeed cover a large number of domains. Here we describe our understanding of its ecosystem, based on our analysis of a large amount of data collected from the domains under leading parking services. Such an understanding also leads to the suspicion of illicit activities within this ecosystem which motivates this research.

Infrastructure. Figure 2 illustrates the infrastructure of the domain parking ecosystem, as revealed by the redirection chains observed during our crawling of parked domains. Those domains are the *start nodes* for the

#	Parking Service	Alexa [2] Global Rank	DailyChanges [34] # Parked Domains	Our data set # Parked Domains	Our data set # Monetization Chains
1	Above	39,901	512,206	17,160	348,534
2	PS1	19,973	1,101,050	11,850	94,794
3	PS2	21,446	1,615,644	9,972	82,258
4	PS3	78,275	261,357	6,447	141,174
5	PS4	59,248	1,275	3,606	174,410
6	PS5	11,357	136,243	2,782	108,956
7	PS6	62,369	330,032	1,020	135,946
8	Rook Media	2,827,350	132,455	562	7,505
9	Fabulous	32,850	406,872	315	53,851
10	InternetTraffic	4,072,159	1,290,732	151	7,764

Table 1: Top 10 Parking services in our data set. The number of monetization chains is not the same as the number of parked domains since each parked domain was crawled multiple times during our collection period. Note that parking services found in this paper to carryout illicit activities are anonymized as "PS#".

whole ecosystem. They include expired domains with back links, blacklisted domains (e.g., exploit servers or TDSes [17]) seized or taken down then repurchased by domain owners or newly acquired domains. As illustrated in the figure, parked domains forward their traffic to parking services, which in turn select the most profitable monetization option in real time, based upon a set of characteristics of the traffic such as its geolocation, browser type and domain keywords. Occasionally, a parking service chooses to forward the traffic to another parking service when the latter offers a higher return on a specific traffic instance. In addition, a parking service may collaborate with *traffic monetization platforms* (e.g. `Skenzo.com`), which monetize different types of traffic such as parking traffic, error traffic (i.e. `404` not found pages) and non-existent domains. Here, we refer to this type of partnership as parking syndication.

The end targets of any traffic monetization option can be either an advertiser, a traffic buyer or a brand name, which are the *end nodes* of the infrastructure.

Potential illicit monetization activities. In our study, we discovered, during our crawling of parked domains, some suspicious activities that call into question the legitimacy of some monetization operations. Specifically, we found that some URLs on the redirection chains initiated by our crawler contain patterns related to the delivery of clicks, for example, "`http://fastonlinefinder.com/ads-clicktrack/click/newjump1.do?`". The problem is that our crawler never clicked on any URLs. It just visited a parked domain and followed its (automatic) redirection chain (see Section 2.3). Also, we observed a lot of "shady" search websites (e.g., `fastonlinefinder.com`), which look like search engines but return low-quality ad results. Those search sites are also observed in prior research [18, 4] and have been presumed to be related to malicious activities like click fraud and malware delivery.

However, confirming the presence of illicit activities in domain monetization is challenging. Take click fraud as an example. We need to determine whether the crawler traffic has indeed been monetized as clicks, which can only be confirmed at the advertiser end. Further complicating this attempt is the observation that some parking services try to make the click delivery look like zeroclick monetization (PPR) by bouncing the traffic through entities with indications of "zeroclick" in the URL: for example, a visit to a parked domain is initially redirected to `http://bodisparking.com/tracking?method=zeroclickrequest` before moving to the click URL. Also, malware scanning cannot find any malicious payloads from the traffic collected from parked domains. Most importantly, given that the traffic for domain monetization goes down a complicated redirection chain, including ad networks and parking-service syndication, it becomes highly nontrivial to identify the party responsible for a malicious activity, even when its presence has been confirmed.

2.3 Overview of Our Study

Here we describe at a high level what we did in our research to understand the suspicious activities that happen within this domain parking ecosystem.

Data collection. As discussed above, the data used in our study was collected from crawling parked domains. For this purpose, we implemented a dynamic crawler as a Firefox extension and deployed it to 29 Virtual Machines (VMs). The crawler is designed to simulate a user's visit to a URL through a browser by rendering its content and running scripts. All such content and HTTP traffic (such as redirections) generated are collected and dumped into a database. In this way, the crawler is able to gather the information produced by execution of dynamic content.

Those crawlers worked on a list of parked domains, which was updated every 3 days during the past 5.5 months (August 1st, 2013 to January 20th, 2014). Those domains were discovered by reverse-lookup for the NS records of known parking services (a list built manually) using the *PassiveDNS* set (DNS record collection) provided by the Security Information Exchange [30]. In order to investigate the monetization activities through those domains, we constructed a *monetization chain* for each URL visit. A monetization chain is a sequence of URL redirections (e.g. HTTP 302, iFrame tags, etc.) observed during a visit to a parked domain, including ad networks and traffic systems related to monetizing the visit.

During each visit, each crawler randomly picked one of 48 user agent strings covering popular browsers, operating systems and mobile devices. Overall, we made about 24M visits to over 100K parked domains. From all those visits, we identified 1.2M (5%) monetization chains including redirections (not direct display of ads).

The leading parking services involved in those chains are presented in Table 1.

Infiltration and expansion. To identify illicit activities involved in the monetization of parked domains and understand the scope and magnitude of the problem, we performed an infiltration study on the domain parking ecosystem to gain an "insider" view about how those parking systems operate. This is critical for overcoming the barriers mentioned in Section 2.2. More specifically, we ran our crawlers to collect data from parked domains and also parked domains under our control with major parking services. Additionally, we launched a few ad campaigns and also purchased traffic associated with some keywords. By carefully selecting the parameters at our discretion, we were able to "connect the dots", linking the start nodes (domains) or traffic sources (crawlers) under our control to our end nodes (ad or traffic purchase campaigns) on monetization chains. Those chains (called *seeds*), together with the accounting information we received from related parking services and ad networks, reveal the whole monetization process with regard to our inputs. This enables us to identify the presence of click fraud, traffic stealing (failing to report monetized traffic) and traffic spam (low-quality traffic). We elaborate this research in Section 3.

To understand the impacts of those fraudulent activities, we extracted from the seed monetization chains a set of fingerprints, or *stamps*, to identify the monetization method used. Once a monetization chain is identified as either PPC or PPR, we infer the presence of illicit activities. Our research shows that our approach accurately identifies illicit monetizations through known ad networks and traffic systems. Most importantly here, this approach helps us *expand* those seeds to a large number of monetization chains collected by our crawlers. Over those chains, we performed a measurement study, which shows the pervasiveness of the problems, their unique features and the profits the parking services get from the illicit activities. The study and its outcomes is reported in Section 4 and Section 5.

Adversary model. In our research, we consider that the parking service is untrustworthy, capable of manipulating the input traffic it receives and its accounting data to maximize its profits at other parties' cost. It also cloaks frequently to avoid being detected by third parties. On the other hand, the service cannot change its interfaces with legitimate ad networks: it needs to make the right calls to deliver its traffic to the networks. In the meantime, some less reputable ad networks (2nd-tier or lower) may not be trustworthy either, which adds complexity to assigning blame to different parties involved in a known fraudulent activity.

In practice, parking services are actually legitimate companies. What we found is that they apparently behave legitimately most of time but are indeed involved in illicit operations occasionally. This adversary is actually very unique, since they blur the lines between fraudulent and legitimate transactions and conduct operations with highly questionable practices.

3 Dark Side of Domain Parking

In this section, we report on our infiltration of the domain parking ecosystem. As discussed before, what we did is to control traffic sources (crawlers), some start nodes (parked domains) and some end nodes (ad campaigns & traffic purchases) of the ecosystem, to get end-to-end monetization chains going through them, as depicted in Figure 3. The figure shows that the chains are as follows: from our parked domains to our end nodes, that is, advertisers or traffic buyers (in black); from other parked domains to our end nodes (in red); from our parked domains to other end nodes (in green) and from our crawlers but not through our domains or end nodes (in blue). Among those chains, the black and red chains connect our traffic source, crawler, to our end nodes through parked domains, which are used as the ground truth for validating our findings (Section 3.3) and the seeds for detecting illicit activities on other chains (Section 4).

Below we describe how we infiltrated the ad networks and direct traffic navigation systems on the end-node side and the parking services on the start-node side.

3.1 Infiltrating End Nodes

Here we walk through our infiltration of the end nodes of the ecosystem, which includes a few steps: we need to identify the right targets (ad networks or traffic systems), register with them, launch ad campaigns and set the right parameters to maximize the chances of receiving our own crawling traffic.

Target identification and registration. To identify the most popular targets, we inspected a sample dataset, including monetization chains collected during the first two weeks of August 2013, to collect a set of the most prevalent top and 2nd-tier ad networks and direct navigation systems. This turned out to be rather straightforward for some targets (e.g., the `Looksmart` ad network with a domain name `looksmart.com`), but not so for others. For example, for some ad networks (e.g. `Advertise`), only the domains of the "shady" search websites they utilized showed up on their click URLs; the "masters" of those search domains were not revealed from their `whois` records, which indicated either an anonymous registration or missing organization names. To uncover those ad networks, what we did include using a domain's Autonomous System Names (ASN) or other domains sharing its IP addresses to determine its affiliation, as well as comparing an ad network's contact

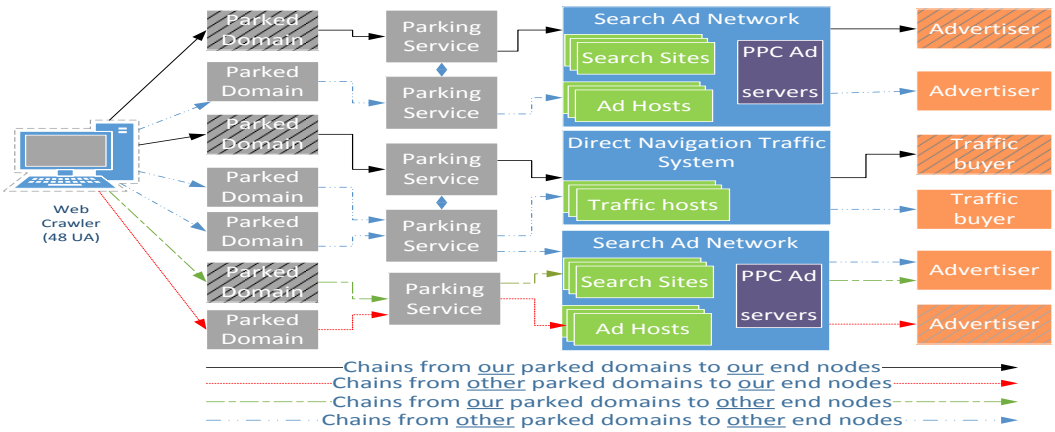


Figure 3: Types of monetization chains captured in our data set where the dashed boxes represent entities under our control.

information with that on the domain's `whois` record.

Once a set of targets (ad networks and traffic systems) were identified, we attempted to register with them as an advertiser or a traffic buyer. This happened with fake identities whenever possible, for the purpose of concealing our true identity to avoid cloaking activities, but we had to use our real information for some of them, which asked for IDs such as driver's license and a credit card. All together, we successfully affiliated ourselves with 15 out of 25 top ad networks and traffic systems identified. For those we failed to do so, the main cause was that they only accepted large-budget customers.

Campaign creation and parameter tuning. We set up a search-like website (Figure 7 in Appendix) and hosted it on three domains, one for traffic purchasing and the other two for advertising. Using those domains, we created both advertising and traffic purchasing campaigns. Specifically, for each of them; we selected 10 keywords related to *our own parked domains*, and also constructed our target URLs, given to the ad networks or traffic systems, to communicate a set of data (e.g., ad network names, publisher IDs, keyword used, etc.) to be used to identify monetization chains that end at our websites through our crawlers. Table 2 summarizes our infiltration meta-data.

For each campaign, we carefully adjusted its parameters to maximize the chances of getting traffic from our own crawlers, which provided us with the end-to-end monetization chains we were looking for. The strategies for such parameter tuning varied across different ad networks and traffic systems. Specifically, some of them offered geo-targeting, which we leveraged to aim at the city our crawlers were located. When this was not available, we tried to take advantage of other features such as browser type and timing when offered. Particularly, for the browser feature, our campaign opted to target the least common browser type, which our crawlers also used for their user agents. Additionally, the timings of some campaigns were tuned in a way that they only ran when traffic from sources other than our crawlers was minimal, for example, from 12AM-6AM and 10PM-12AM. As an example, the direct navigation traffic system `DNTX` provided only country-level geo-location and the platforms (iOS, Android, BlackBerry and others) under its mobile/tablet category. In this case, we targeted our campaigns at the country our crawlers were running from and the Blackberry platforms.

Overall, we ran advertising and traffic campaigns through all 15 ad and traffic networks starting from Nov. 22nd last year, which cost us $2,260 in total. The logistics of our end node infiltration campaigns are summarized in Table 3. Among those campaigns, the two with `Admanage` lasted for only two days before we were locked out from our account for almost a month without giving any explanations. Also, our campaign with `Approved Search` did not receive any traffic that fit our targeting criteria.

3.2 Infiltrating Start Nodes

To infiltrate start nodes, we went through top 10 most popular parking services identified from our sample dataset (Section 3.1), opened accounts with them and carefully chose our monetization options so as to maximize the chance of observing illicit activities.

Parking service registration. Among all the services we tried, `Domain Power` asked for real identity information, which we skipped, and `trafficZ` turned down our application, citing the small volume of traffic our domains received. Other parking services performed some type of authentication, such as sending a PIN number to a valid phone number, and verifying the consistency between the owner information on our domains' `whois` records and that on our application. We passed those checks and used fake identities to register with 7 parking

Parked domains	FREE-JOBS.INFO, JOBS-BOARD.INFO, REAL-JOBS.INFO, NEWS-CHANNEL.BIZ, NEWS-FEED.BIZ, FAMILY-VACATION.ORG DREAM-VACATION.ORG, COUPONS-FREE.INFO, LOCAL-COUPONS.INFO, SUPERCOUPONS.INFO, CLOTHES-SHOP.INFO DESIGNER-CLOTHES.INFO, TEAMXYZ.INFO, XYZAGENT.INFO, LOWCOST-FLOWERS.COM, EDITING-SOFTWARE.ORG EDUCATION-RESOURCES.ORG, EDUCATION-GUIDE.ORG, MARKETING-EDUCATION.ORG, CITY-CARS.NET, MUSIC-LIVE.ORG SOFWTARE.COM, NEWS-NETWORK.BIZ
Target Keywords	Jobs, Cars, News, Vacation, Coupons, Clothes, Software, Education, Music, Flowers
Target URL example	`http://anwers.net/search.php?s=advertise&c=camp7&type=kw&kw=jobs&aff=63567&geo=us_in_bloomington`

Table 2: Infiltration meta-data. The URL example indicates the ad network is "Advertise", click keyword is "Jobs" and the publisher ID is "63567".

Network	Campaign Type	Total Hits	AVG CPC/CPR	Budget ($)	# Days	Targeting
adMarketplace	PPC	143	0.2	100.13	18	City & Time
Advertise	PPC	18	0.067	125.17	17	City, Device & Time
Google AdWords	PPC	1	1.86	222.71	32	City & Device
Affinity$^+$	PPC	371	0.3	250.36	17	City & Time
Approved Search	PPC	0	0.05	10.5	2	Country & Time
Bidvertiser	PPC	0	0.1	160.13	28	Country
Bing Ads$^+$	PPC	6	0.97	33.06	30	City, Device, Time
Ezanga	PPC	35	0.21	47.92	33	City & Time
Looksmart	PPC	131	0.19	91.1	30	City & Time
Avenue5	PPC	0	0.05	44.25	37	Country & Time
Admanage	PPC	0	0.4	43.6	2	Country
7search$^+$	PPC	3	0.1	297	26	Country, Device
	PPR	146	0.1	203	29	& Time
DNTX	PPC	1	0.058	155.89	13	Country
	PPR	29	0.056	203.5	27	& Device
Adspark	PPR	106	0.2	31.8	5	City, Device & Time
Trellian	PPR	25	0.201	250	43	Country

Table 3: End node Infiltration Logistics. $^+$ Networks that required registration with a real identity. "Total hits" represent click/traffic hits received at our web server and initiated by our crawler. Note that `7Search` offered both PPC and PPR campaign types and as such, we created two campaigns with it, one for each type.

services as illustrated in Table 4. Most of these services are indeed popular as shown earlier in Table 1.

#	Parking Service	# Parked Domains
1	PS5	9
2	PS2	2
3	PS6	6
4	PS1	4
5	PS4	3
6	Rook Media	2
7	PS7	2

Table 4: Parking Infiltration Logistics. Note that the total number of domains does not add up to 23 which is due to moving some domains between parking services. Note that PS7 is not shown in Table 1 as it is not in the top 10 parking services.

Domain monetization. We purchased 23 domains under a number of top level domains and parked them with the 7 services. The names of those domains were carefully chosen to match the keywords we targeted in our campaigns (see Table 2). We also set their NS records to point to those of their corresponding parking services, with some exceptions discussed later.

Our preliminary analysis on data crawled from parking services showed that suspicious activities were only observed on redirection chains. Therefore, we tried to avoid situations like PPC listings, where a set of PPC ads are displayed on a parking landing page. Instead, we chose not to have such a page (displaying signs like "for sale"), so the parking services can monetize the traffic our domains received through redirections. Actually, one parking service, `Bodis`, allowed us to explicitly set this monetization option by redirecting the traffic instead of setting the NS record to `Bodis`. For example, we parked the domain `news-network.biz` with `Bodis` by using the GoDaddy forwarding service to send our domain traffic to `http://bodisparking.com/news-network.biz`.

3.3 Findings

Through infiltrating the start and end nodes of the ecosystem and crawling domains hosted by popular parking services, we were able to collect 1,015 monetization chains that link our crawlers to our end nodes (our advertising or traffic purchase websites), sometimes through our start nodes (parked domains). Using those chains as the ground truth, we confirmed the presence of illicit activities during parked domain monetization, including click fraud, traffic spam and traffic stealing, as elaborated below.

Click fraud. Out of all the ad clicks delivered to our advertising websites through parking services, 709 were found to come from our own crawlers. They are clearly fraudulent since our crawlers were designed not to click on any ad (Section 2.3). Table 3 details the number of clicks received from our crawler through each ad network.

Traffic spam. The 4 traffic purchasing campaigns we launched received 306 traffic hits from our crawler through domains parked with parking services. Upon examining the parked domains that served as the start nodes on those monetization chains, we found that 83 of them were totally unrelated to the keywords we purchased from the direct traffic systems. Table 5 provides examples of spam and good-quality traffic.

Keyword	Spam	Not Spam
Music	19jj.com, ib2c.com.cn	thepiatebay.org, itunesstore.de
Software	almacenyhostpublico.com	linuxfab.cx,iphoneos3.com
Coupon	seattleseoforum.com	coupons-free.info
"Others"	brf.no, betovilla.com,gddfg.com	education-guide.org
	70263.com, facebooki.pl	dolla.com

Table 5: "Spam" and "Not Spam" examples of parked domains appearing in traffic purchase monetization chains leading to our traffic buyer website. "Others" represents instances where the purchased keyword was not propagated through to our traffic buyer URL but it is still evident from the spam examples that they are not related to any of the 10 keywords we purchased.

Traffic stealing. Occasionally, parking services were found to be dishonest with domain owners, failing to inform them for part of the revenue they were supposed to share with the owners. Specifically, we cross-examined the revenues of the domains under our control and the billing reports for the ad/traffic campaigns we launched. This revealed that some monetization chains going through our parked domains were not reported to us (domain owners) but charged to our campaign accounts.

For example, we confirmed the existence of traffic stealing from monetization chains captured by our crawler connecting three of our domains (parked with PS5) and our PPR campaign with `7Search`. This was achieved through comparing the billing reports provided by `7Search`, the parked domains' revenue reports provided by the parking service and the related monetization chains with the right combination of time stamp, source IP address, referral domain and keyword. It turns out that we, as a campaign owner, were billed for 23 traffic hits by `7Search` (see Figure 6(b) in Appendix) but nothing was reported by the parking service (see Figure 6(a) in Appendix) in December 2013. We show the breakdown of the crawlers' traffic in Table 6. Clearly, the parking services kept the rightful share away from us as the owners of the parked domains. Note that not all requests from our crawler were billed by `7Search` because they limit the traffic hits by one IP address and a valid visiting period (our campaign was set to run between 12AM-6AM and 10PM-11:59PM). Additionally, we found other monetization chains, captured by our crawler and monetized by the same parking service through other ad networks such as `Advertise` that have not been reported on our parked domains' revenue reports.

		Traffic Reported by	
Parked Domain	Crawler	Parking Service	Billed by 7search
Coupons-free.info	24	0	16
Real-jobs.info	23	0	5
News-feed.info	21	0	2

Table 6: Traffic stealing through 3 of our parked domains in the month of December, 2013.

4 Fingerprinting Monetization Chains

Through the infiltration study, we confirmed the presence of illicit activities in the monetization of parked domains. What is less clear, however, is the pervasiveness and impact of those activities. Understanding of this issue cannot rely on the 1,015 seed chains (reported in Table 3) whose traffic sources and end nodes were under our control. We need to identify the illicit operations occurring on other monetization chains, particularly those blue and green ones in Figure 3, which do not connect to our ad/traffic campaigns. To this end, we developed a technique that fingerprints the monetization options observed on our seed chains. These fingerprints, which we call *stamps*, were used to "expand" the seed set, capturing the illicit activities on other monetization chains collected by our crawlers over months.

4.1 Methodology

The idea. As discussed above, the problem of detecting illegitimate operations, which we did not have a direct observation of, comes down to identifying the monetization options they involve. More specifically, as soon as we know exactly how a parking service monetizes a visit from our crawler, we can immediately find out whether a fraudulent activity occurred: clearly, the PPC option is a fraudulent click, as our crawler never clicked; when it comes to PPR, we check the consistency between the keywords expected by the end nodes and the names of the parked domains the traffic went through (see Section 5 for details). Therefore, the question here becomes how to determine which options have been used in a given monetization chain.

Actually, even though those options might not be evident on a monetization chain, we know that it must go through a corresponding monetization party (ad networks, traffic systems, etc.) before the traffic gets to the end node. This needs to happen for accounting purposes: for example, if a click has not been sent to a PPC ad network, the ad network never knows about it and therefore will not be able to pay its publisher or bill its advertiser. Also, the last few URLs leading to the end node are clearly related to the monetization party. As an example, let us look at a monetization chain captured by our crawler in Table 7, which was initiated by a visit to a domain parked with `PS1` and ended at our advertiser site. Looking backward from our URL along the chain, we can see two URLs from `fastonlinefinder.com`, a search website. The site turns out to be affiliated with the `Advertise` ad network. Interestingly, once we compare this path with other chains also through the same ad network, it becomes quite clear that they carry a unique pattern: first, right before the end node, the last two URLs are always similar; second, for these two URLs, even though they vary across different chains in terms of search websites, the remaining part mostly stays constant. This observation was further verified by the click URLs for sponsored ads from the same ad network, which we obtained by registering with `Advertise` as a publisher (see Appendix for details on search sites and sponsored click URLs).

The above example shows that we can leverage the ordered sequence of URL patterns to determine the presence of a monetization option. Such a sequence is a "stamp" we utilize to expand our seed set to find other illicit monetization chains within the dataset collected by our crawlers. Following we describe the methodology

#	URL	Description
1,2	http://bastak-taraneh.com/	Parked domain
3	http://otnnetwork.net/?epl=...	Parking service anchor
4,5	http://67.201.62.155/index2.html?q=...&des=	AdLux
6	http://21735.1b2a3r4w5dgp6v.filter.nf.adlux.com/ncp/checkbrowser?key...	ad network (syndicate)
7	**http://fastonlinefinder.com/ads-clicktrack/click/newjump1.do?...&terms=ticketsoftware...**	Search site for
8	**http://fastonlinefinder.com/ads-clicktrack/click/newjump2.do?terms=ticketsoftware&...**	Advertise ad network
9	http://anwers.net/search.php?s=advertise...&kw=software...	Our Advertiser

Table 7: End-to-End monetization chain example of a visit to a parked domain leading to our advertiser page. For clarity sake, we omit parts of the URLs.

for clustering and generalizing URLs from a monetization party, and extracting the stamps from the URL patterns.

URL-IP Cluster (UIC) generation. A specific URL of a monetization party (ad network, traffic systems, etc.) can be too specific for fingerprinting its monetization activity. An ad network can have many affiliated websites and each site may have multiple domains and IP addresses. To utilize such a URL for generating a stamp, we first need to generalize it across those domains, addresses and potential variations in its file path and other parameters. To this end, we clustered related URLs into *URL-IP clusters* (UIC). A UIC includes a set of IP addresses for related hosts and the invariant part of the URL (without the host name) across all members in the cluster. The former describes the ownership of this set of URLs and the latter represents their common functionality, which together fingerprints a monetization option with regard to an ad network or a traffic system.

To cluster a group of URLs into UICs, we first extracted the host part of a URL, replacing it with all IPs of the domain, and then broke the remaining part of the URL into tokens. A token is either the full path of the URL including file name (which is typically very short for a monetization URL) or an argument. The value of the argument was removed, as it can be too specific (e.g. keyword and publisher ID). Over those IP-token sets, we ran a clustering algorithm based upon Jaccard indices for both IPs and tokens, as follows:

1. Each URL (including an IP set and a token set) is first assigned to a unique UIC.

2. Two UICs are merged together when *both* their IP sets and token sets are close enough (Jaccard indices above their corresponding thresholds).

3. Repeat step 2 until no more UICs can be merged.

A pair of thresholds are used here to determine the similarity of two IP sets (T_{ip}) and two token sets (T_{tok}) respectively. In our research, we set T_{ip} to 0.1 and T_{tok} to 0.5, and ran the algorithm to cluster all the URLs on the 1.2M monetization chains collected by our crawlers. By replacing individual URLs with their corresponding UICs, we obtained 429K unique generalized chains, which were further used to detect illicit activities.

Stamp extraction. The stamps for different monetization options were extracted from seed monetization chains, after generalizing them using the aforementioned UICs. Specifically, we utilized 715 UIC chains (generalized from the 1,015 chains reported in Table 3) to fingerprint 11 ad networks and traffic systems, with one stamp created for all the campaigns associated with a given ad network or traffic system. For this purpose, we applied a 2-fold cross-validation approach to generate stamps and assess their effectiveness. Specifically, we randomly split the UIC chains for each campaign into two equal sized sets, one for stamp extraction (training) and the other for stamp evaluation (test). Over the training set, we determined a stamp by traversing each UIC chain backwards and selecting the sequence of UICs shared by *all* the chains involving a certain ad network or traffic system. Typically the longest sequence identified in this way became the stamp for all the chains going through its related monetization organization. However, for a campaign with a small number of UIC chains (e.g. `BingAds`), we only utilized the last common UIC (right before our advertiser's URL) across all the chains as the stamp.

All together, our approach generated stamps for 11 ad networks and traffic systems. Ad network stamps contained on average two UICs while traffic system stamps were mostly one UIC in length.

4.2 Evaluation

False negative. Using all the 11 stamps generated from the training set, we analyzed all the monetization chains within the test set. For each campaign, its stamp was found to match all of its monetization chains in the test set and thus no false negative was observed.

False positives. We further evaluated the false positive rate that could be introduced by the stamps on a dataset containing 768M redirection chains collected by crawling the top 1M Alexa websites [2] Jan 21-31, 2014. The purpose here is to understand whether a redirection chain not involving clicks or traffic selling can be misidentified as a related monetization chain and whether the monetization chains of one ad network or traffic system can be classified as those of another party. By applying our stamps on the dataset, we flagged 12 chains as matches to click stamps and another 19 chains as matches to traf-

fic stamps. Upon manually analyzing the 12 click chains, we found that all of them were actually fraudulent clicks generated by parking services (10 chains) and a traffic/ad network named CPX24 (2 chains). CPX24 in our case generated clicks on its own hosted ads when traffic from publishers flowed in. For the 19 traffic chains, they were indeed PPR monetization chains. Further, by manually searching for a set of ad network specific domain names such as `affinity.com` within the Alexa dataset, we discovered a number of redirection chains going through the same ad networks fingerprinted by our stamps but *not* involving any clicks (e.g. ad display and conversion tracking URLs). None of them were misidentified by our stamps as click-based monetization chains.

Discussion. Our evaluation shows that the stamps generated over UICs accurately capture all monetization chains associated with a specific ad network or traffic system. However, those stamps are designed for analyzing the traffic through individual organizations, which is enough for our purpose of understanding the scope and magnitude of fraudulent activities, not for detecting those operations on any monetization chains, particularly those belonging to other monetization parties. Also note that we cannot use existing ad-blocking lists such as EasyList [26] to serve our goal, due to its limitations: first, EasyList does not distinguish between a click and an impression (ad display); second, search websites used by ad networks to deliver clicks are not covered by the block list; finally, the list fails to include the hosts and URLs of traffic monetization systems.

5 Measurements

In this section, we report our measurement study on illicit parked domain monetization. This study is based upon a dataset of 1.2M monetization chains collected in a 5.5-month span. Such data were first labeled using the "stamps" generated (Section 4.1) from the seed data to identify the monetization options associated with individual chains, and then analyzed to understand the pervasiveness of illicit monetization practices and its financial impact. Here we elaborate on the outcomes of this study.

5.1 Dataset Labeling

Expansion. To perform the measurement study, we labeled the 1.2M monetization chains collected from crawling parked domains by "expanding" the 1,015 seed chains (the ground truth) to this much larger dataset. Specifically, we generated UICs over those 1.2M chains, generalized the seed chains using those UICs, and then extracted click and traffic stamps from the seeds as described in Section 4.1. Matching those stamps to the generalized UIC chains in the larger dataset (429K UIC chains), we were able to label 120,290 (28.03%) UIC chains corresponding to 212,359 (17.1%) URL monetization chains. The labeled set includes two monetization options, PPC (45.7%) and PPR (56.3%) where 2% of them include both PPR and PPC monetizations on the same chain as explained later.

Unknown set. Although many chains were successfully labeled, there are almost 308K UIC chains in the dataset not carrying any stamps, which were marked as "unknown". Looking into this unknown set through random sampling, we found that it exhibited consistent patterns related to click delivery and traffic selling which can be added to the labeled set if we had verified seed chains. For example, we found many other ad-nets such as `Adknowledge` and `Bidvertiser` (2.9%), and other traffic monetization systems such as `Adrenalads` and `ZeroRedirect` (19.5%). Particularly, `Sendori`, a traffic platform, is widely present, covering 5.4% of the chains in the dataset. 2.6% of those chains actually led to domain name marketplaces such as `SnapNames`, and a large portion (over 19.7%) of them stop at some parking services and traffic monetization platforms (e.g. `Skenzo`) with error messages indicating cloaking behavior.

5.2 Monetization Decisions

Over the labeled dataset, we analyzed the parties responsible for such decisions and the way those decisions were made, based on a categorization of the parked domains involved.

Monetization decision maker. Finding the party that chooses a monetization option is important, as it tells us who is the ultimate culprit for an illicit activity. However, this is challenging, due to the syndication of multiple monetization parties, among parking services, ad-nets and traffic systems. Within our dataset, we found that these types of syndications are pervasive (49.5%). As an example, `AdLux` in Table 7 is actually a syndicate of `Advertise`, displaying its ads and sharing its click revenue. In the presence of a syndication, a starting node's parking service may not be responsible for the follow-up illicit monetization, which could actually be performed by one of its syndicates. To this end, we identify the parking service of the starting node to be the responsible party of a monetization chain only when the click or traffic stamp appears right after the starting node (i.e. parked domain). When there are other entities between the parked domain and the stamp, we use a *parking-service anchor* as described below.

Typically, a parking service funnels the traffic from its parked domains to a "controller" domain, which we call a parking service (PS) anchor, for choosing a monetization option. Our idea here is to locate the PS anchor right before a click or traffic stamp. When this happens, the owner of the anchor is clearly responsible for

the monetization decision. To this end, we picked out the most prevalent second UICs down individual monetization chains (which is expected to cover over 50% of all the chains associated with a specific parking service), and identified its ownership using its `whois` records and Name Server. Such a UIC is considered to be an anchor for the parking service.

In our research, we identified anchors for 4 of the most prevalent parking services in our set. Some parking services such as `PS6` and `PS4` launder all the traffic through direct navigation traffic systems (`DNTX` and `ZeroRedirect` respectively) which are owned by the parent companies of the two parking services. Since those traffic systems are used by other clients, we did not consider them to be the anchors of the parking service. Using the heuristic described earlier (i.e. direct link from a parked domain to a stamp) and the list of anchors, we assigned each monetization chain to the parking service responsible for the selection of its monetization option, as illustrated in Table 8. Here the "unknown" category includes the chains we could not determine the parties responsible for their monetizations, due to the disconnection between the parked domain or PS anchor and the click or traffic stamp, with unknown UICs standing in-between. For example, the chain in Table 7 was marked as "unknown", as the known anchor `http://otnnetwork.net/?epl=` is separated from the ad stamp there.

Impacts of domain categorization. As discussed before, our research focuses on the redirection chains generated by 1.2M out of 24M visits to parked domains. The rest of those visits only resulted in a simple display of PPC ads, which were less likely to be used for illicit monetization. The fact that those redirection chains were so rare to see here can be attributed to IP cloaking. In the meantime, we believe that this is also caused by the way that traffic from different domains is monetized. Specifically, a parking service like `Bodis` often classifies domains into "primary" or "secondary". Primary domains are those accepted by top-tier search networks (e.g. `Google AdWords`) to display their ads while secondary ones are less trusted, including those serving malicious content before taken down and the ones related to typos of trade or brand names. The secondary domains here are much more likely to lead to redirection chains, as discovered in our research (illustrated by Figure 4). Among all domains visited by our crawlers, we found (using [6]) that only 2.9% of them were considered to be secondary, which naturally limits the number of the redirection chains we could observe.

5.3 Illicit Monetization

In this section, we report our findings about the prevalence of illicit monetization practices, particularly click

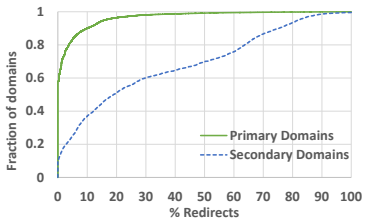


Figure 4: Tendency towards redirect monetization of "secondary" domains. "% Redirects" represents the percentage of redirection chains observed per parked domain.

		PS5	PS6	PS1	PS2	Others	Unknown
PPC Ad-nets	adMarketplace	1		9,526	1670	4	35,704
	Advertise	10,277		2,329		730	3,576
	Affinity						9,904
	Google AdWords						18
	Bing Ads						18,186
	Looksmart					385	143
	Ezanga					3	422
PPR Traffic Systems	7Search -Spam	7,766				666	1,484
	7Search - Malware	43					
	DNTX - Spam	758	21,959	189	2	25,776	2,305
	DNTX - Malware	45	3,217	9	1	2,924	113
	Trellian - Spam	1		9	1	11,781	
	Trellian - Malware					1	
	AdsPark -Spam		1,755				2,183
	AdsPark - Malware						1
Totals		18,803 (11.02%)	22,425 (13.15%)	13,808 (8.1%)	1,673 (1%)	39,872 (23.4%)	73,949 (43.4%)

Table 8: Illicit activities observed by parking services in the labeled set. "Others" refers to some of the parking services shown in Table 1 not necessarily anonymized.

fraud, traffic spam and the malware distribution discovered during our analysis of the labeled dataset. Note that we did not measure traffic stealing, as this activity could only be observed on the monetization chains whose start and end nodes were under our control.

Traffic spam. Using the traffic stamps, we discovered 119K (56.3%) traffic monetization chains. To identify the presence of traffic spam on each of those chains, we compared the keywords associated with its start node domain with those of its end node (assuming that end nodes purchased keywords related to the contents of their domains). This works as follows:

• *Keyword generation*. To generate keywords for both the start and end nodes, we used a keyword suggestion tool by `BingAds` [1], a tool widely used by advertisers to select keywords for ad targeting. This tool automatically created a list of keywords (including typos) for each domain (start and end nodes).

• *Keyword filtering*. At this step, we cleaned the list of keywords, discarding common ones ("www", "com", etc.). Specifically, we calculated the normalized entropy of each word as the prior work did [31] and then removed the 50 words with the lowest entropy (i.e. highly popular). Also dropped from the keyword list were determiners, pronouns, interjection and "wh"-words ("what", "where", etc.), which are unlikely to be related to specific

domain content. For this purpose, we filtered out the keywords using *Stanford CoreNLP* [35], a natural language processing tool for part-of-speech tagging and stemming.

• *Keyword matching*. Comparing the keywords of the start node (i.e., parked domain) with those of the end node (traffic purchase website) on a monetization chain, we considered the chain to be traffic spam if individual keywords of its parked domain did not match *any* words associated with its end node. In the case that one of these two domains do not have any keywords, we attempted to match the other domain's keywords to its domain name. If this attempt succeeds, the chain would not be considered as spam, otherwise; it would.

As a result, we found that 70.7% of all PPR monetization chains are traffic spam as illustrated in Table 8 and attributed to each parking service and traffic system. Table 9 provides some traffic spam examples received by popular brand names.

End node	Parked domain examples
Amazon.com	craigslits.com, 14.de, audii.de
Apple.com	acgeo.com, backlinkscenter.info
Coupons.com	4google.com,agendo.com
Sears.com	uasairways.com, cursoblogger.com
Expedia.com	pizzahutjobs.com, financetasksforce.com

Table 9: Examples of end nodes receiving spam traffic.

Click fraud. All labeled 97K (45.7%) PPC monetization chains are clearly fraudulent clicks, as our crawlers never clicked on any ads. Table 8 provides a breakdown of fraudulent clicks observed from each parking service through ad-nets for which we have a click stamp. By taking a close look at the ad-nets involved, we found that none of the fraudulent clicks on the top-tier networks (`Google AdWords` & `BingAds`) could be attributed to a parking service, due to ad-net syndications. Parking services avoid clicking on top-tier ad-nets' ads because they have a better click fraud detection system than 2nd-tier networks and as such they only happen through ad-net syndication. Additionally, 2% of the fraudulent clicks could not be attributed to a parking service due to the presence of a traffic stamp between the start node (i.e parked domain) and the click stamp. For example, domains parked with `PS6` resulted in fraudulent clicks through a traffic system (`DNTX`) which is owned by the same parent company of `PS6`, namely `TeamInternet AG`.

Also interesting is the observation that not only were the clicks delivered through those chains completely fraudulent but they often came from parked domains that had nothing to do with the ad campaigns at the end nodes. Specifically, we applied the keyword generation and matching approach described above to analyze the relations between the parked domains on those chains and their corresponding end nodes. This study reveals that 61.3% of the fraudulent clicks were from parked domains completely unrelated to the end nodes on their chains. Also given the fact that the average cost-per-click (CPC), which is $0.28, is twice as much as the average cost-per-redirect (CPR) that we paid, there is no legitimacy whatsoever in such click-faking activities.

(a) Revenue Estimates

	P_{FPPC}	P_{PPR}	P_{TS}	P_{PPC}	P_{PPA}	$\frac{Rev_{Fraud}}{Rev}$
PS5	0.01	0.01	0.78	0.97	0.0057	40.3%
PS1	0.0015	0.0003	0.77	0.998	0.00003	7.4%
PS3	0.0004	0.004	0.71	0.995	0.0001	9.3%
PS2	0.0001	0.0000004	0.7	0.9997	0.00016	0.8%
PS6	0	0.015	0.66	0.983	0.0015	18.5%
PS4	0	0.0073	0.64	0.976	0.017	10%

(b) Description of variables used.

Legitimate	P_{PPC}	Probaility of monetization through the display of Pay-Per-Click (PPC) ads.
	P_{PPR}	Probability of monetization through Pay-Per-Redirect (PPR).
	P_{PPA}	Probaility of monetization through affiliate marketing, Pay-Per-Action (PPA).
Fraudulent	P_{FPPC}	Probability of monetization through a fraudulent click on a Pay-Per-Click (PPC) ad.
	P_{TS}	Probaility of monetization through traffic spam in Pay-Per-Redirect (PPR).

Table 10: Estimates of illicit monetization revenues for selected parking services.

Malware distribution. Also discovered in our research is parking services' involvement (probably unwittingly) in malware distribution. We found that many PPR monetization chains were leading to malicious content, either through drive-by downloads or through social engineering scams such as FakeAV or flash player updates (see Figure 8 in Appendix). This occurred because the traffic systems involved did not do their due diligence in detecting the traffic buyers who actually disseminate malware. Using content structure clustering, a technique applied by prior research [13], we concluded that *at least* 3.7% of the PPR traffic buyers spread malware. This illicit activity not only hurts the victims visiting a parked domain but also affects the parked domain when it gets blacklisted by URL scanners such as SafeBrowsing [12], which reduces the monetary value of the parked domain when its owner decides to sell it.

5.4 Revenue Analysis

Model. As discussed before, parking services are unique in that their monetization operations involve both legitimate and illegitimate activities. To understand the economic motives behind this monetization strategy, we analyzed their revenues with a model derived from that used in prior research [23]:

$$Rev = Visits \cdot (Rev_{Fraud} + Rev_{Legit})$$

where the total revenue Rev is calculated from the total number of visits and the average revenue for each visit. This average revenue is further broken down into two components, the part from illicit monetization (Rev_{Fraud}) and that from legitimate monetization (Rev_{Legit}). These components were further estimated as follows:

$$Rev_{Fraud} = P_{FPPC} \cdot CPC + P_{PPR} \cdot CPR \cdot P_{TS}$$
$$Rev_{Legit} = P_{PPC} \cdot p_{pclk} \cdot CPC + P_{PPR} \cdot CPR \cdot P_{\widetilde{TS}} + P_{PPA} \cdot p_{pact} \cdot CPA$$

Intuitively, the above two equations describe five possible situations when a visit to a parked domain is monetized: illicit activities (click fraud or traffic spam) or legitimate ones (legitimate click, direct traffic or affiliate marking monetization). The revenue component from the illicit activities is estimated using the probability of click fraud P_{FPPC} and that of traffic spam $P_{PPR} \cdot P_{TS}$, where P_{PPR} is the probability of direct traffic monetization (PPR) and P_{TS} is the chance of traffic spam when PPR is chosen, together with the revenues for a click *CPC* and a redirect *CPR*. Similarly, the legitimate revenue component comes from a PPC display (with a probability P_{PPC}) given that a user clicks on one of the ads (with a click-through rate p_{pclk}), type-in traffic (P_{PPR}) when it is *not* subject to traffic spam (with a probability $P_{\widetilde{TS}}$), or affiliate marketing (P_{PPA}) when the user performs the operation expected (with a probability p_{pact}). The revenues for those legitimate activities are *CPC*, *CPR* and *CPA* respectively.

Results. In our analysis, we estimated all the probabilities above (P_{FPPC}, P_{PPR}, P_{TS}, P_{PPC} and P_{PPA}) using the *larger* set of all 24M visits to 100K parked domains in a 5.5-month span (Section 2.3). Also, the click-through rate p_{pclk} and a user's probability of taking an action under PPA were both set to 0.02, and *CPA* to $0.265, all according to the prior work [23], while CPC and CPR were determined as $0.28 and $0.14 respectively, based on the average cost for our ad/traffic campaigns.

In the absence of data about the total number of visits per parking service, all we could do is estimate the portion of its income from the illicit activities to its total revenue, based upon our data. The results are shown in Table 10.

Discussion. From the table, we can see that even reputable parking services like `PS2` have at least 0.8% of its revenue come from illicit monetizations. For others, this revenue source is even more significant (e.g., 40.3% for `PS5` whom we found to be aggressive in its illicit monetizations). Revenue from fraudulent clicks is found to be zero for `PS6` and `PS4` because, as described earlier in Section 5.3, they are bouncing their traffic through their own traffic systems and as such we can not attribute fraudulent clicks to them. Note that our estimates here are very conservative, due to the cloaking those services played to our crawlers (which used a small set of IP addresses) and the limited scope of our study (which only covered 11 ad-nets and traffic systems). We expect that the ratios of illicit revenues are much higher in practice.

6 Discussion

Domain Parking Regulation. Our study uncovers the illicit monetizations by parking services but the underlying problem is even graver. Currently, there is no regulation on the behaviors of parking services, which allows them to set up arbitrary terms of services accommodating their own benefits. It is worth noting that parking services may have started to exhibit illicit monetization activities due to the decline in their revenues [3, 29]. Also, our research shows they have a tendency to profit from secondary domains illicitly (Figure 4), due to the difficulty in monetizing those domains through a legitimate channel. Protecting the advertisers' and traffic buyers' benefits in the existence of dishonest parking services is challenging because incoming traffic can be manipulated. What complicates the situation more is that ad-nets could be owned by the company who also runs parking services and the advertisers have no fair party to talk to. Further, direct navigation traffic (i.e. zeroclick) is being advocated by parking services and there is no guarantee on the quality of the incoming traffic. Our research discloses the dark side of parked domain monetization which calls for serious policy efforts to regulate parking services.

Here, we suggest several practices that could mitigate many types of illicit monetization activities when enforced. First, the advertisers should be provided with a clearer picture of the monetization activities. For example, the types of publishers should be marked out as well to advertisers besides their publishers' IDs, which helps advertisers in auditing and monitoring traffic coming from parking services. In fact, some ad-nets are already moving to such direction: for example, Affinity, a popular ad-net, distinguishes publishers by assigning them types such as "in-text" and "domain zero click". We also suggest providing traffic buyers with a way to check the integrity of incoming traffic such as passing the domain name of each start node in the referral. Enforcement and compliance of such mechanisms requires the presence of a 3rd party service (i.e. policy enforcer) in the ecosystem.

Legal and ethical concerns. There are several ethical concerns raised during our study, and we carefully designed our experiments to address them. First, we crawled our own parked domains which is problematic if we earn profit from it. To address this issue, we avoid cashing in the revenues we earned from the 7 parking services hosting our parked domains ($81.06 in total). Second, we ignore *robots.txt* served by parking services when crawling since we focused on their illicit behaviors. Other studies on malicious activities also ignore the *robots.txt* file [17, 33, 19, 7]. Third, one may question that the artificial traffic generated by our crawler

could affect the advertisers or traffic buyers. In fact, we crawled parked domains in a moderate speed and parking services have deployed mechanisms to discern artificial traffic and stop charging the advertisers when identified [28]. Lastly, we ran campaigns with ad-nets and traffic systems but there was no actual business running. The websites and advertisements we set up were coherent to the policies of ad-nets and traffic systems. There was no damage to visitors and we did not collect any Personally Identifiable Information (PII) from them.

7 Related Work

Parking services. Although domain parking services have been here for years, little has been done to understand their security implications. What comes close are the works on typo-squatting [37, 22, 9], which reveals that domain owners utilize this technique for profit. Also, prior research shows that malicious domains tend to be parked once detected [27, 17]. Most related to our research is the study on click spam [10], which focuses on click-spam detection and also mentions the possible involvement of one parking service (`Sedo`) in such activities based on its JavaScript. Such code was not found in our research. Compared with the prior work, what we did is a systematic study on the illicit activities of parking services, which has never been done before. This is made possible by the new infiltration analysis we performed. Our study not only confirms the presence of illicit operations within parking services but also brings to light their scope and magnitude.

Illicit activities in online advertising. Ad-related illicit activities have been extensively studied. Examples include click-fraud [21, 4, 5, 11, 25], drive-by-download [18], trending-term exploitation [23] and impression fraud [32]. Such prior work all looks at a conventional adversary who performs malicious activities whenever possible. The parking services, however, are very different: they run legitimate business with advertisers and ad networks. However, our study reveals that a significant portion of their revenues actually come from illicit activities, which raises the awareness about this completely unregulated business.

Infiltration into malicious infrastructure. To understand how underground businesses work, a lot of studies attempt to infiltrate their business infrastructure. Examples include the work on Spam [16, 15], CAPTCHA solving [24], blackhat SEO [36] and Pay-per-install networks [8]. Different from such prior research, we need to infiltrate the parking monetization process without disrupting its operations. This was achieved using a new approach through which we controlled some nodes on both ends of the monetization ecosystem and managed to link them together.

8 Conclusion

This paper reports the first systematic study on illicit activities in parked domain monetization. To demystify this "dark side" of parking services, we devised an infiltration analysis to gain control of some start nodes and end nodes of the parking ecosystem, and then connect the dots, sending our crawling traffic across the nodes under our control on the both ends, with the monetization entities (domain parking services, ad networks) in-between. This analysis provided us a unique observation of the whole monetization process, which enabled us to confirm the presence of click fraud, traffic spam and traffic stealing. We further expanded those seed chains to millions of monetization chains collected over 5.5 months, using the stamps of their monetization options. Over such data, our study revealed the pervasiveness of the illicit monetization practices and their revenues, which calls for policy efforts to control those illicit operations.

Acknowledgements

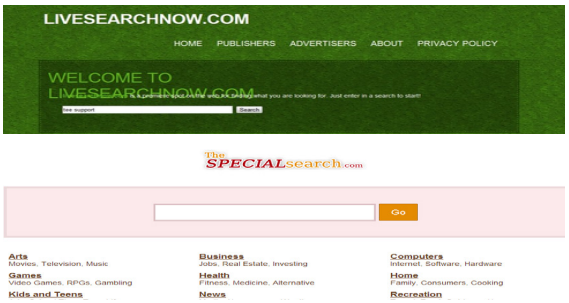
We thank our shepherd Damon McCoy and anonymous reviewers for their insightful comments and suggestions. We thank Yinglian Xie and Fang Yu from Microsoft Research for their valuable comments. This work is supported in part by NSF CNS-1017782, 1117106, 1223477 and 1223495. Any opinions, findings, conclusions or recommendations expressed in this paper do not necessarily reflect the views of the NSF.

Appendix

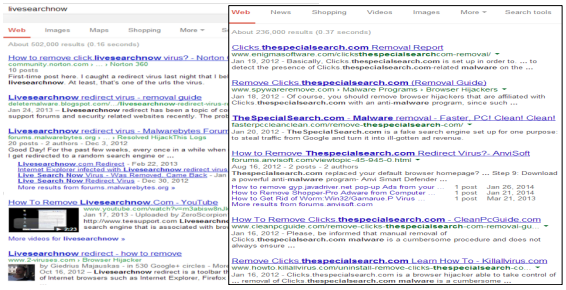
The true operation of the shady search sites. `fastonlinefinder.com` is one of a large number of search sites we refer to as "shady". They are shady in that they rarely display organic results and emphasize on sponsored ads. Moreover, they have been reported in previous works on click fraud [18, 4] and have been presumed malicious to some extent. Additionally, many victims have often complained about a "redirect" malware hijacking their traffic and redirecting to these search sites as shown in Figure 5.

Through our empirical investigations, we discovered the actual role they play. We found their true operation was to act as click servers for search ads (similar to traditional click servers of other none search advertisements) and as such they are owned and operated by ad-nets. Another use of those search sites was to set the click referral and as such, the advertiser will assume their ad was displayed on the referring search site.

It is important to note here, that the use of such search sites is not illegal. It is only misunderstood due to their abuse by ad-net publishers. A fraudulent publisher will use a malware or Trojan to generate clicks on their ads and since the clicks lead to an ad-net's search sites, the



(a) Screen shots of two ad-net search sites: `livesearchnow.com` with Advertise & `thespecialsearch.com` with Affinity.

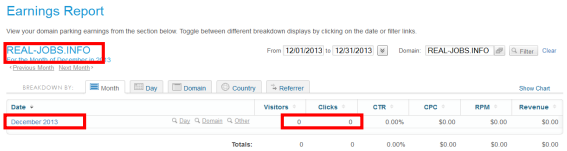


(b) Google search results showing malware complaints for the two search sites.

Figure 5: Ad-net search site examples showing screen shots of the search sites and malware related complaints by users.

search site become wrongly accused as the malicious party.

Evidence Survey. Along our infiltration we collect a set of evidence to support our findings. We start with Figure 6 which confirms traffic stealing by one parking service that is not reporting traffic, as shown in 6(a), which have been monetized through `7search` and verified by our payment for the traffic as shown in 6(b).



(a) Screen shot of our parked domain revenue report with the parking service in question.

CLICK DATE (GMT-06:00)	REFERRING DOMAIN	IP	KEYWORD	CPC
12/9/2013 21:46	http://**********/REAL-JOBS.INFO?	Crawler IP	cf job	$0.10
12/11/2013 21:00	http://**********/REAL-JOBS.INFO?	Crawler IP	cf job	$0.10
12/12/2013 1:05	http://**********/REAL-JOBS.INFO?	Crawler IP	cf job	$0.10
12/12/2013 23:26	http://**********/REAL-JOBS.INFO?	Crawler IP	cf job	$0.10
12/13/2013 21:52	http://**********/REAL-JOBS.INFO?	Crawler IP	cf job	$0.10

(b) Billing report by 7Search shows 5 billed traffic hits from our parked domain. Part of the referral is removed to anonymize the parking service.

Figure 6: Traffic stealing observed on our parked domain

Additionally, We verify the association of search sites to ad-nets by registering with two ad-nets (`Advertise` & `Bidvertiser`) as a publisher interested in displaying their sponsored ads. We set up our website with a search service that pulls organic search results from `Google` and sponsored ads from the two ad-nets we registered with. By pulling sponsored ads from the ad-nets, we verified the use of search sites as the click URLs as shown in Figure 7 which shows one click URL by `Advertise` that has the same URL tokens as the URLs in Table 7. Actually, the same website, `toppagefinder.com`, appeared also in our data set and as such was in the same UIC which was a correct association. Note that the same website used here for our publisher was also used for our advertising and traffic buying campaigns.

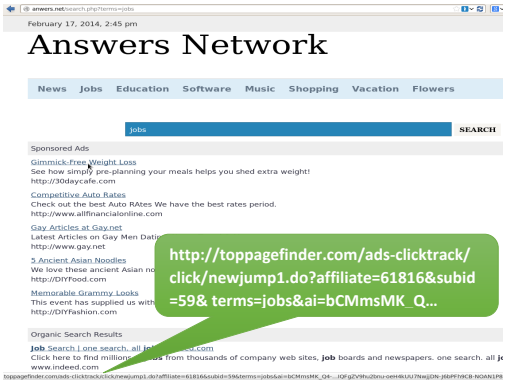


Figure 7: Our Advertiser, publisher and traffic buyer website.

Finally, in Figure 8 we show examples of visits to domains parked with `PS5` leading to malware downloads through two traffic systems, namely `DNTX` and `ZeroRedirect`.

References

[1] ADS, B. Bing ads api. `https://developers.bingads.microsoft.com/`.

[2] ALEXA. Alexa top global sites. `http://www.alexa.com/topsites`, February 2014.

[3] ALLEMANN, A. Sedo reports continuing decline in domain parking. `https://domainnamewire.com/2013/11/12/sedo-reports-continuing-decline-in-domain-parking/`, November 2013.

[4] ALRWAIS, S. A., GERBER, A., DUNN, C. W., SPATSCHECK, O., GUPTA, M., AND OSTERWEIL, E. Dissecting ghost clicks: Ad fraud via misdirected human clicks. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 21–30.

[5] BLIZARD, T., AND LIVIC, N. Click-fraud monetizing malware: A survey and case study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on* (Oct 2012), pp. 67–72.

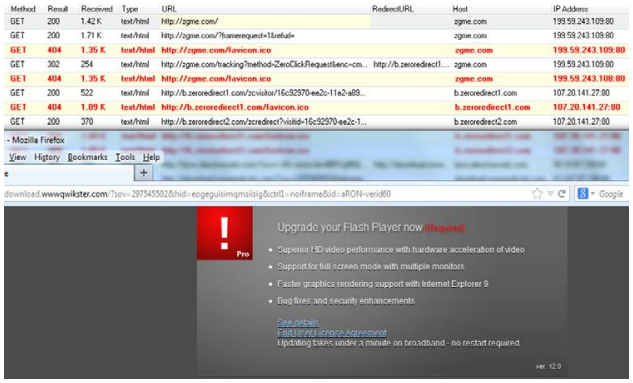
[6] BODIS. Javascript and xml api. `https://www.bodis.com/news/javascript-and-xml-api`.

[7] BORGOLTE, K., KRUEGEL, C., AND VIGNA, G. Delta: Automatic Identification of Unknown Web-based Infection Campaigns. In *Proceedings of the ACM Conference on Computer and Communications Security* (2013), CCS '13, ACM.

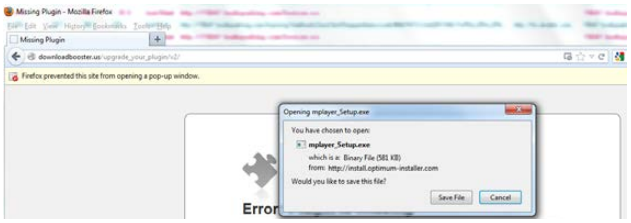
[8] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 13–13.

[9] COULL, S., WHITE, A., YEN, T.-F., MONROSE, F., AND REITER, M. Understanding domain registration abuses. In *Security and Privacy Silver Linings in the Cloud*, K. Rannenberg, V. Varadharajan, and C. Weber, Eds., vol. 330 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2010, pp. 68–79.

[10] DAVE, V., GUHA, S., AND ZHANG, Y. Measuring and fingerprinting click-spam in ad networks. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug.



(a) Screen shot of a visit to a domain parked with PS5 leading to a flash player update malware scam through the ZeroRedirect traffic system. A snapshot of the HTTP traffic is also shown which illustrates the redirection process.



(b) Screen shot of a visit to a domain parked with PS5 leading to malware download through the DNTX traffic system

Figure 8: Visits to parked domains leading to malware distribution.

2012), 175–186.

[11] DAVE, V., GUHA, S., AND ZHANG, Y. Viceroi: Catching click-spam in search ad networks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 765–776.

[12] GOOGLE. Safe browsing api google developers. `https://developers.google.com/safe-browsing/`.

[13] HACHENBERG, C., AND GOTTRON, T. Locality sensitive hashing for scalable structural classification and clustering of web documents. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management* (New York, NY, USA, 2013), CIKM '13, ACM, pp. 359–368.

[14] HUANG, W. Parked domain numbers and traffic, and more on the exploits served. `http://blog.armorize.com/2010/08/parked-domain-numbers-and-traffic-and.html`, August 2010.

[15] KANICH, C., WEAVERY, N., MCCOY, D., HALVORSON, T., KREIBICHY, C., LEVCHENKO, K., PAXSON, V., VOELKER, G. M., AND SAVAGE, S. Show me the money: characterizing spam-advertised revenue. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 15–15.

[16] LEVCHENKO, K., CHACHRA, N., ENRIGHT, B., FELEGYHAZI, M., GRIER, C., HALVORSON, T., KANICH, C., KREIBICH, C., LIU, H., MCCOY, D., PITSILLIDIS, A., WEAVER, N., PAXSON, V., VOELKER, G. M., AND SAVAGE, S. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proceedings of 32nd annual Symposium on Security and Privacy* (May 2011), IEEE.

[17] LI, Z., ALRWAIS, S., XIE, Y., YU, F., AND WANG, X. Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 112–126.

[18] LI, Z., ZHANG, K., XIE, Y., YU, F., AND WANG, X. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 674–686.

[19] LU, L., PERDISCI, R., AND LEE, W. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 467–476.

[20] MAHJOUB, D. A look at the relationship between parked domains and malware. `http://labs.umbrella.com/2013/03/20/discovery-of-new-suspicious-domains-using-authoritative-dns-traffic-and-parked-domains-analysis/`, March 2013.

[21] MILLER, B., PEARCE, P., GRIER, C., KREIBICH, C., AND PAXSON, V. What's clicking what? techniques and innovations of today's clickbots. In *Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment* (Berlin, Heidelberg, 2011), DIMVA'11, Springer-Verlag, pp. 164–183.

[22] MOORE, T., AND EDELMAN, B. Measuring the perpetrators and funders of typosquatting. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security* (Berlin, Heidelberg, 2010), FC'10, Springer-Verlag, pp. 175–191.

[23] MOORE, T., LEONTIADIS, N., AND CHRISTIN, N. Fashion crimes: Trending-term exploitation on the web. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 455–466.

[24] MOTOYAMA, M., LEVCHENKO, K., KANICH, C., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. Re: Captchas: Understanding captcha-solving services in an economic context. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 28–28.

[25] PEARCE, P., GRIER, C., PAXSON, V., DAVE, V., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. The zeroaccess auto-clicking and search-hijacking click fraud modules. Tech. Rep. UCB/EECS-2013-211, EECS Department, University of California, Berkeley, Dec 2013.

[26] PETNEL, R. Easylist. `https://easylist-downloads.adblockplus.org/easylist.txt`.

[27] RAHBARINIA, B., PERDISCI, R., ANTONAKAKIS, M., AND DAGON, D. Sinkminer: Mining botnet sinkholes for fun and profit. In *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, 2013), USENIX.

[28] SEDO. Domain parking terms and conditions. `https://sedo.com/us/about-us/policies/domain-parking-terms-and-conditions-sedocom/?tracked=1&partnerid=38758&language=us`.

[29] SEDO HOLDING. Sedo holding ag 6-month report. `http://www.sedoholding.com/fileadmin/user_upload/Dokumente/English/Reports_2013/Sedo_Holding_6M_Report_2013.pdf`, 2013.

[30] SIE, I. Security information exchange (sie) portal. `https://sie.isc.org/`.

[31] SINKA, M. P., AND CORNE, D. W. Towards modernised and web-specific stoplists for web document analysis. In *Web Intelligence, 2003. WI 2003. Proceedings. IEEE/WIC International Conference on* (2003), IEEE, pp. 396–402.

[32] SPRINGBORN, K., AND BARFORD, P. Impression fraud in online advertising via pay-per-view networks. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 211–226.

[33] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 133–144.

[34] TOOLS, D. Daily dns changes and web hosting activity. `http://www.dailychanges.com/`, February 2014.

[35] TOUTANOVA, K., KLEIN, D., MANNING, C. D., AND SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1* (2003), Association for Computational Linguistics, pp. 173–180.

[36] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Juice: A longitudinal study of an seo botnet. In *NDSS* (2013), The Internet Society.

[37] WANG, Y.-M., BECK, D., WANG, J., VERBOWSKI, C., AND DANIELS, B. Strider typo-patrol: Discovery and analysis of systematic typo-squatting. In *Proceedings of the 2Nd Conference on Steps to Reducing Unwanted Traffic on the Internet - Volume 2* (Berkeley, CA, USA, 2006), SRUTI'06, USENIX Association, pp. 5–5.

Towards Detecting Anomalous User Behavior in Online Social Networks

Bimal Viswanath, *MPI-SWS*
M. Ahmad Bashir, *MPI-SWS*
Mark Crovella, *Boston University*
Saikat Guha, *MSR India*
Krishna P. Gummadi, *MPI-SWS*
Balachander Krishnamurthy, *AT&T Labs–Research*
Alan Mislove, *Northeastern University*

Abstract

Users increasingly rely on crowdsourced information, such as reviews on Yelp and Amazon, and liked posts and ads on Facebook. This has led to a market for black-hat promotion techniques via fake (e.g., Sybil) and compromised accounts, and collusion networks. Existing approaches to detect such behavior relies mostly on supervised (or semi-supervised) learning over known (or hypothesized) attacks. They are unable to detect attacks missed by the operator while labeling, or when the attacker changes strategy.

We propose using unsupervised anomaly detection techniques over user behavior to distinguish potentially bad behavior from normal behavior. We present a technique based on Principal Component Analysis (PCA) that models the behavior of normal users accurately and identifies significant deviations from it as anomalous. We experimentally validate that normal user behavior (e.g., categories of Facebook pages liked by a user, rate of like activity, etc.) is contained within a low-dimensional subspace amenable to the PCA technique. We demonstrate the practicality and effectiveness of our approach using extensive ground-truth data from Facebook: we successfully detect diverse attacker strategies—fake, compromised, and colluding Facebook identities—with no *a priori* labeling while maintaining low false-positive rates. Finally, we apply our approach to detect click-spam in Facebook ads and find that a surprisingly large fraction of clicks are from anomalous users.

1 Introduction

The black-market economy for purchasing Facebook likes,[1] Twitter followers, and Yelp and Amazon reviews has been widely acknowledged in both industry and academia [6, 27, 37, 58, 59]. Customers of these black-market services seek to influence the otherwise "organic" user interactions on the service. They do so through a variety of constantly-evolving strategies including fake (e.g., Sybil) accounts, compromised accounts where malware on an unsuspecting user's computer clicks likes or posts reviews without the user's knowledge [35], and incentivized collusion networks where users are paid to post content through their account [7, 8].

When (if) an attack is detected, the affected service usually takes corrective action which may include suspending the identities involved in the attack or nullifying the impact of their attack by removing their activity in the service. One approach for defense used today by sites like Facebook is to raise the barrier for creating fake accounts (by using CAPTCHAs or requiring phone verification). However, attackers try to evade these schemes by using malicious crowdsourcing services that exploit the differences in the value of human time in different countries. Another approach used widely today is to detect misbehaving users after they join the service by analyzing their behavior. Techniques used to address this problem to date have focused primarily on detecting specific attack strategies, for example, detecting Sybil accounts [10, 65, 67], or detecting coordinated posting of content [36]. These methods operate by assuming a particular attacker model (e.g., the attacker is unable to form many social links with normal users) or else they train on known examples of attack traffic, and find other instances of the same attack. Unfortunately, these approaches are not effective against an adaptive attacker. It is known that attackers evolve by changing their strategy, e.g., using compromised accounts with legitimate social links instead of fake accounts [14, 15, 35], to avoid detection.

In this paper we investigate a different approach: detecting *anomalous* user behavior that deviates significantly from that of normal users. Our key insight, which we validate empirically, is that normal user behavior in online social networks can be modeled using

[1] When printed in this font, likes refer to Facebook "Like"s (i.e., the action of clicking on a Like button in Facebook).

only a small number of suitably chosen latent features. Principal Component Analysis (PCA), a technique with well-known applications in uncovering network traffic anomalies [44], can be used to uncover anomalous behavior. Such anomalous behavior may then be subjected to stricter requirements or manual investigations.

We make the following three contributions: First, we introduce the idea of using PCA-based anomaly detection of user behavior in online social networks. PCA-based anomaly detection requires that user behavior be captured in a small number of dimensions. As discussed in more detail in Section 4, using over two years of complete user behavior data from nearly 14K Facebook users, 92K Yelp users, and 100K Twitter users (all sampled uniformly at random), we find that the behavior of normal users on these social networks can be captured in the top three to five principal components. Anomalous behavior, then, is user behavior that cannot be adequately captured by these components. Note that unlike prior proposals, *we do not require labeled data in training the detector*. We train our anomaly detector on a (uniformly) random sampling of Facebook users which contains some (initially unknown) fraction of users with anomalous behavior. Using PCA we are able to distill a detector from this unlabeled data as long as a predominant fraction of users exhibit normal behavior, a property which is known to hold for Facebook.

Second, we evaluate the accuracy of our PCA-based anomaly detection technique on ground-truth data for a diverse set of normal and anomalous user behavior on Facebook. To do so, we acquired traffic from multiple black-market services, identified compromised users, and obtained users who are part of incentivized collusion networks. Our approach detects over 66% of these misbehaving users at less than 0.3% false positive rate. In fact, the detected misbehaving users account for a large fraction, 94% of total misbehavior (number of likes). Section 6 reports on the detailed evaluation.

Lastly, in Section 7 we apply our technique to detect anomalous ad clicks on the Facebook ad platform. Where only 3% of randomly sampled Facebook users had behavior flagged by us as anomalous (consistent with Facebook's claims [32]), a significantly higher fraction of users liking our Facebook ads had behavior flagged as anomalous. Upon further investigation we find that the like activity behavior of these users is indistinguishable from the behavior of black-market users and compromised users we acquired in the earlier experiment. Our data thus suggests that while the fraction of fake, compromised or otherwise suspicious users on Facebook may be low, they may account for a disproportionately high fraction of ad clicks.

2 Overview

Our goal is to detect anomalous user behavior without *a priori* knowledge of the attacker strategy. Our central premise is that attacker behavior should appear anomalous relative to normal user behavior along some (unknown) latent features. Principal Component Analysis (PCA) is a statistical technique to find these latent features. Section 3 describes PCA and our anomaly-detection technique in detail. In this section we first build intuition on why attacker behavior may appear anomalous relative to normal user behavior (regardless of the specific attacker strategy), and overview our approach.

2.1 Illustrative Example and Intuition

Consider a black-market service that has sold a large number of Facebook likes in some time frame to a customer (e.g., the customer's page will receive 10K likes within a week). Since a Facebook user can contribute at most one like to a given page, the black-market service needs to orchestrate likes from a large number of accounts. Given the overhead in acquiring an account—maintaining a fake account or compromising a real account—the service can amortize this overhead by selling to a large number of customers and leveraging each account multiple times, once for each customer. Such behavior may manifest along one of two axes: temporal or spatial (or both). By *temporal* we mean that the timing of the like may be anomalous (e.g., the inter-like delay may be shorter than that of normal users, or the weekday-weekend distribution may differ from normal). By *spatial* anomaly we mean other (non-temporal) characteristics of the like may be anomalous (e.g., the distribution of page categories liked may be different, or combinations of page categories rarely liked together by normal users may be disproportionately more frequent).

A smart attacker would attempt to appear normal along as many features as possible. However, each feature along which he must constrain his behavior reduces the amortization effect, thus limiting the scale at which he can operate. We show in Section 6 that black-market users we purchased have nearly an order of magnitude larger number of likes than normal users, and four times larger number of categories liked. If the attacker constrained himself to match normal users, he would require significantly more accounts to maintain the same level of service, adversely affecting profitability.

In the above illustrative example, it is not clear that the number of likes and categories liked are the best features to use (in fact, in section 6.4 we show that such simple approaches are not very effective in practice). Some other feature (or combination of features) that is even more discriminating between normal and anomalous behavior and more constraining for the attacker may be bet-

ter. Assuming we find such a feature, hard-coding that feature into the anomaly detection algorithm is undesirable in case "normal" user behavior changes. Thus, our approach must automatically find the most discriminating features to use from unlabeled data.

2.2 Approach

At a high level, we build a model for normal user behavior; any users that do not fit the model are flagged as anomalous. We do not make any assumptions about attacker strategy. We use PCA to identify features (dimensions) that best explain the predominant normal user behavior. PCA does so by projecting high-dimensional data into a low-dimensional subspace (called the *normal subspace*) of the top-N principal components that accounts for as much variability in the data as possible. The projection onto the remaining components (called the *residual subspace*) captures anomalies and noise in the data.

To distinguish between anomalies and noise, we compute bounds on the L^2 norm [43] in the residual subspace such that an operator-specified fraction of the *unlabeled* training data (containing predominantly normal user behavior) is within the bound. Note that the normal users do not need to be explicitly identified in the input dataset. When testing for anomalies, any data point whose L^2 norm in the residual subspace exceeds the bound is flagged as anomalous.

2.3 Features

We now discuss the input features to PCA that we use to capture user behavior in online social networks. We focus on modeling Facebook like activity behavior and describe suitable features that capture this behavior.

Temporal Features: We define a temporal feature as a *time-series of observed values*. The granularity of the time-series, and the nature of the observed value, depends on the application. In this paper, we use the number of likes at a per-day granularity. In general, however, the observed value may be the time-series of number of posts, comments, chat messages, or other user behavior that misbehaving users are suspected of engaging in.

Each time-bucket is a separate dimension. Thus, for a month-long trace, the user's like behavior is described by a ∼30-dimensional vector. The principal components chosen by PCA from this input set can model inter-like delay (i.e., periods with no likes), weekday-weekend patterns, the rate of change of like activity, and other latent features that are linear combinations of the input features, without us having to explicitly identify them.

Spatial Features: We define a spatial feature as a *histogram of observed values*. The histogram buckets depend on the application. In this paper, we use the category of Facebook pages (e.g., sports, politics, education) as buckets, and number of likes in each category as the observed value. In general, one might define histogram buckets for any attribute (e.g., the number of words in comments, the number of friends tagged in photos posted, page-rank of websites shared in posts, etc).

As with temporal features, each spatial histogram bucket is a separate dimension. We use the page categories specified by Facebook[2] to build the spatial feature vector describing the user's like behavior, which PCA then reduces into a low-dimensional representation.

Spatio-Temporal Features: Spatio-temporal features combine the above two features into a single feature, which captures the *evolution of the spatial distribution of observed values* over time. In essence, it is a time-series of values, where the value in each time bucket summarizes the spatial distribution of observed values at that time. In this paper, we use *entropy* to summarize the distribution of like categories. Entropy is a measure of information content, computed as $-\sum_i p_i \log_2 p_i$, where bucket i has probability p_i. In general, one might use other metrics depending on the application.

Multiple Features: Finally, we note that temporal, spatial, and spatio-temporal features over multiple kinds of user behavior can be combined by simply adding them as extra dimensions. For instance, like activity described using l_T temporal dimensions, l_S spatial dimensions, and l_{ST} spatio-temporal dimensions, and wall posting activity described similarly (p_T, p_S, p_{ST}), can be aggregated into a vector with $\sum_x l_x + \sum_x p_x$ dimensions passed as input into PCA.

3 Principal Component Analysis (PCA)

Principal component analysis is a tool for finding patterns in high-dimensional data. For a set of m users and n dimensions, we arrange our data in an $m \times n$ matrix $\mathbf{X}$, whose rows correspond to users and whose columns correspond to user behavior features discussed above. PCA then extracts common patterns from the rows of $\mathbf{X}$ in an optimal manner. These common patterns are called *principal components*, and their optimality property is as follows: over the set of all unit vectors having n elements, the first principal component is the one that captures the *maximum variation* contained in the rows of $\mathbf{X}$. More formally, the first principal component v_1 is given by:

$$v_1 = \arg\max_{||v||=1} ||\mathbf{X}v||.$$

The expression $\mathbf{X}v$ yields the inner product (here, equivalent to the correlation) of v with each row of $\mathbf{X}$; so v_1

[2] Facebook associates a topic category to each Facebook page which serves as the category of the like.

maximizes the sum of the squared correlations. Loosely, v_1 can be interpreted as the n-dimensional pattern that is most prevalent in the data. In analogous fashion, for each k, the k^{th} principal component captures the maximum amount of correlation beyond what is captured by the previous $k-1$ principal components.

The principal components $v_1, \ldots, v_n$ are constructed to form a *basis* for the rows of $\mathbf{X}$. That is, each row of $\mathbf{X}$ can be expressed as a linear combination of the set of principal components. For any principal component v_k, the amount of variation in the data it captures is given by the corresponding *singular value* σ_k.

A key property often present in matrices that represent measurement data is that only a small subset of principal components suffice to capture most of the variation in the rows of $\mathbf{X}$. If a small subset of singular values are much larger than the rest, we say that the matrix has *low effective dimension.* Consider the case where r singular values $\sigma_1, \ldots, \sigma_r$ are significantly larger than the rest. Then we know that each row of $\mathbf{X}$ can be approximated as a linear combination of the first r principal components $v_1, \ldots, v_r$; that is, $\mathbf{X}$ has *effective dimension* r.

Low effective dimension frequently occurs in measurement data. It corresponds to the observation that the number of factors that determine or describe measured data is not extremely large. For example, in the case of human-generated data, although data items (users) may be described as points in high-dimensional space (corresponding to the number of time bins or categories), in reality, the set of factors that determine typical human behavior is not nearly so large. A typical example is the user-movie ranking data used in the Netflix prize; while the data matrix of rankings is of size about 550K users $\times$ 18K movies, reasonable results were obtained by treating the matrix as having an effective rank of 20 [41]. In the next section, we demonstrate that this property also holds for user behavior in online social networks.

4 Dimensioning OSN User Behavior

To understand dimensionality of user behavior in online social networks, we analyze a large random sampling of users from three sources: Facebook, Yelp, and Twitter. The Facebook data is new in this study, while the Yelp and Twitter datasets were repurposed for this study from [50] and [4] respectively. We find low-effective dimension in each dataset as discussed below.

4.1 User Behavior Datasets

We use Facebook's people directory [25] to sample Facebook users uniformly at random.[3] The directory summarizes the number of people whose names start with a given character `x`, and allows direct access to the y^{th} user with name starting with `x` at `https://www.facebook.com/directory/people/x-y`. We sample uniformly at random from all possible (1.14B) `x-y` pairs, and follow a series of links to the corresponding user's profile.

We collected the publicly visible like and Timeline [34] activity of 13,991 users over the 26 month period ending in August 2013. For each user, we record three types of features: (i) *temporal*, a time-series of the number of likes at day granularity resulting in 181 dimensions for a 6-month window, (ii) *spatial*, a histogram of the number of likes in the 224 categories defined by Facebook, and (iii) *spatio-temporal*, a time-series of entropy of like categories at day granularity (181 dimensions for 6 months). We compute the entropy H_t on day t as follows: for a user who performs n_t^i likes in category i on day t, and n_t likes in total on day t, we compute $H_t = -\sum_i \frac{n_t^i}{n_t} \log_2 \frac{n_t^i}{n_t}$.

The Yelp dataset consists of all 92,725 Yelp reviewers in the San Francisco area [50] who joined before January 2010 and were active (wrote at least one review) between January 2010 and January 2012. The spatial features are constructed by a histogram of number of reviews posted by the user across 445 random groupings of 22,250 businesses[4] and 8 additional features (related to user reputation provided by Yelp[5]). The dataset also contains temporal features, the time-series of the number of reviews posted by a user at day granularity resulting in 731 dimensions covering the two year period.

The Twitter dataset consists of a random sample of 100K out of the 19M Twitter users who joined before August 2009 [4]. Previous work [4] identified topical experts in Twitter and the topics of interests of users were inferred (e.g., technology, fashion, health, etc) by analyzing the profile of topical experts *followed* by users. In this dataset, each expert's profile is associated with a set of topics of expertise. We construct a spatial histogram by randomly grouping multiple topics (34,334 of them) into 687 topic-groups and counting the number of experts a user is following in a given topic-group. The Twitter dataset does not have temporal features.

4.2 Low-Dimensionality of User Behavior

A key observation in our results from all three online social networks (Facebook, Yelp, Twitter) across the three user behaviors (temporal, spatial, and spatio-temporal)

[3] Users may opt-out of this directory listing. However, our analysis found 1.14 billion users listed in the directory as of April 2013, while Facebook reported a user count of 1.23 billion in December 2013 [31]. We therefore believe the directory to be substantially complete and representative.

[4] Randomly grouping the feature space helps compress the matrix without affecting the dimensionality of the data [13].

[5] Examples of reputation features include features such as number of review endorsements and number of fans.

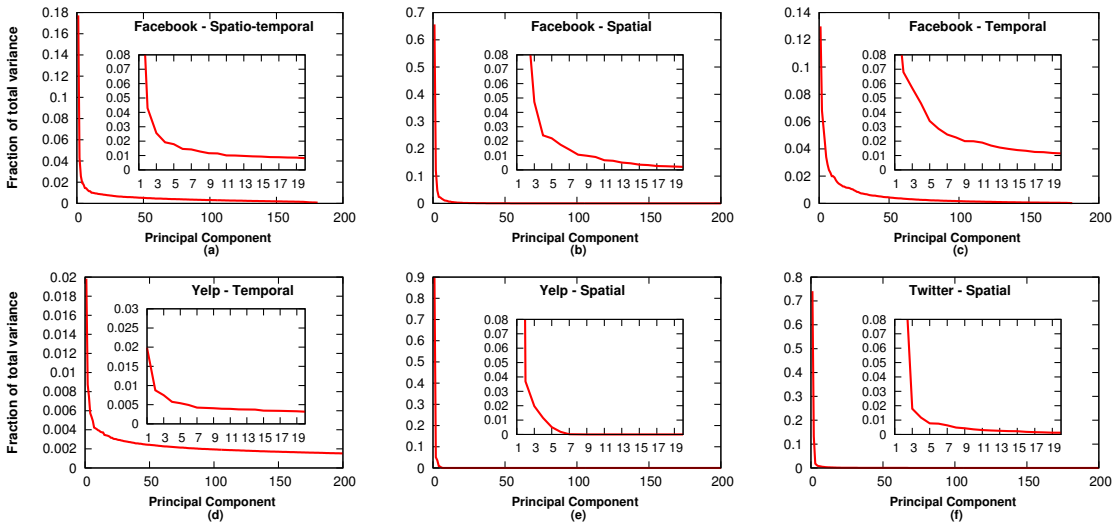


Figure 1: **Scree plots showing low-dimensionality of normal user behavior. A significant part of variations can be captured using the top three to five principal components (the "knee" of the curves).**

is that they all have low effective dimension. Figure 1 presents *scree plots* that show how much each principal component contributes when used to approximate the user behavior matrix $\mathbf{X}$, and so gives an indication of the effective dimension of $\mathbf{X}$. The effective dimension is the x-value at the "knee" of the curve (more clearly visible in the inset plot that zooms into the lower dimensions), and the fraction of the area under the curve left of the knee is the total variance of the data accounted for. In other words, the important components are the ones where the slope of the line is very steep, and the components are less important when the slope becomes flat. This method of visually inspecting the scree plot to infer the effective dimension is known as Cattell's Scree test in the statistics literature [5].

For Facebook like behavior (Figure 1(a)–(c)), the knee is around five principal components. In fact, for spatial features in Facebook like activity (Figure 1(b)), these top five components account for more than 85% of the variance in the data. We perform a parameter sweep in Section 6 and find that our anomaly detector is not overly sensitive (detection rate and false positives do not change drastically) to minor variations in the choice of number of principal components [54]. Yelp and Twitter (Figure 1(d)–(f)) show a knee between three and five dimensions as well. Overall, across all these datasets where the input dimensionality for user behavior were between 181 and 687, we find that the effective dimensionality is around three to five dimensions.

5 Detecting Anomalous User Behavior

In this section, we elaborate on the normal subspace and residual subspace discussed in Section 2, and describe how an operator can use them to detect anomalous behavior.

The operation of separating a user's behavior into principal components can be expressed as a *projection* . Recall that the space spanned by the top k principal components $v_1, \ldots, v_k$ is called the *normal subspace.* The span of the remaining dimensions is referred to as the *residual subspace.* To separate a user's behavior, we project it onto each of these subspaces. Formulating the projection operation computationally is particularly simple since the principal components are unit-norm vectors. We construct the $n \times k$ matrix $\mathbf{P}$ consisting of the (column) vectors $v_1, \ldots, v_k$. For a particular user's behavior vector x, the normal portion is given by $x_n = \mathbf{P}\mathbf{P}^T x$ and the residual portion is given by $x_r = x - x_n$.

The intuition behind the *residual subspace detection* method for detecting anomalies is that if a user's behavior has a large component that cannot be described in terms of *most* user's behavior, it is anomalous. Specifically, if $\|x_r\|_2$ is unusually large where $\|\cdot\|_2$ represents the L^2 norm, then x is likely anomalous. This requires setting thresholds for $\|x_r\|_2^2$ known as the squared prediction error or SPE [44]. We discuss how we choose a threshold in Section 6.

5.1 Deployment

In practice, we envision our scheme being deployed by the social network operator (e.g., Facebook), who has

access to all historical user behavior information. The provider first selects a time window in the past (e.g., $T =$ 6 months) and a large random sample of users active during that time (e.g., 1M) whose behavior will be used to train the detector. As described earlier, training involves extracting the top k principal components that define the normal and residual subspace for these users. This training is repeated periodically (e.g., every six months) to account for changes in normal user behavior.

The service provider detects anomalous users periodically (e.g., daily or weekly) by constructing the vector of user behavior over the previous T months, projecting it onto the residual subspace from the (latest) training phase, and analyzing the L^2 norm as discussed earlier. Since each user is classified independently, classification can be trivially parallelized.

6 Evaluation

We now evaluate the effectiveness of our anomaly detection technique using real-world ground-truth data about normal and anomalous user behavior on Facebook. Our goal with anomaly detection in this section is to detect Facebook like spammers.

6.1 Anomalous User Ground Truth

We collected data for three types of anomalous behaviors: fake (Sybil) accounts that do not have any normal user activity, compromised accounts where the attacker's anomalous activity interleaves with the user's normal activity, and collusion networks where users collectively engage in undesirable behavior. We used the methods described below to collect data for over 6.8K users. We then used Selenium to crawl the publicly visible data for these users, covering 2.16M publicly-visible likes and an additional 1.19M publicly-visible Timeline posts including messages, URLs, and photos. We acquired all activity data for these users from their join date until end of August 2013.

Black-Market Services: We searched on Google for websites offering paid Facebook likes (query: "buy facebook likes"). We signed up with six services among the top search results and purchased the (standard) package for 1,000 likes; we paid on average $27 to each service. We created a separate Facebook page for each service to like so we could track their performance. Four of the services [18–21] delivered on their promise (3,437 total users), while the other two [22, 23] did not result in any likes despite successful payment.

As mentioned, we crawled the publicly-visible user behavior of the black-market users who liked our pages. We discovered 1,555,534 likes (with timestamps at day granularity) by these users. We further crawled the users' publicly visible Timeline for public posts yielding an additional 89,452 Timeline posts.

Collusion Networks: We discovered collaborative services [7, 8] where users can collaborate (or collude) to boost each other's likes. Users on these services earn virtual credits for liking Facebook pages posted by other users. Users can then "encash" these credits for likes on their own Facebook page. Users can also buy credits (using real money) which they can then encash for likes on their page. We obtained 2,259 likes on three Facebook pages we created, obtaining a set of 2,210 users, at an average cost of around $25 for 1,000 likes. The price for each like (in virtual credits) is set by the user requesting likes; the higher the price, the more likely it is that other users will accept the offer. We started getting likes within one minute of posting (as compared to more than a day for black-market services).

As with black-market users, we crawled the user activity of the users we found through collusion networks. We collected 359,848 likes and 186,474 Timeline posts.

Compromised Accounts: We leveraged the browser malware Febipos.A [35] that infects the user's browser and (silently) performs actions on Facebook and Twitter using the credentials/cookies stored in the browser. The malware consists of a browser plugin, written in (obfuscated) Javascript, for all three major browsers: Chrome, Firefox and Internet Explorer [28, 29].

We installed the malware in a sandbox and deobfuscated and analyzed the code. The malware periodically contacts a CnC (command-and-control) server for commands, and executes them. We identified 9 commands supported by the version of the malware we analyzed: (1) like a Facebook page, (2) add comments to a Facebook post, (3) share a wall post or photo album, (4) join a Facebook event or Facebook group, (5) post to the user's wall, (6) add comments to photos, (7) send Facebook chat messages, (8) follow a Twitter user, and (9) inject third-party ads into the user's Facebook page.

We reverse-engineered the application-level protocol between the browser component and the CnC server, which uses HTTP as a transport. We then used `curl` to periodically contact the CnC to fetch the commands the CnC would have sent, logging the commands every 5 minutes. In so doing, we believe we were able to monitor the entire activity of the malware for the time we measured it (August 21–30, 2013).

Identifying which other Facebook users are compromised by Febipos.A requires additional data. Unlike in the black-market services and collusion networks—where we were able to create Facebook pages and give to the service to like— we can only passively monitor the malware and cannot inject our page for the other infected users to like (since we do not control the CnC server).

To identify other Facebook users compromised by

Febipos.A, we identified two commands issued during the week we monitored the malware: one which instructed the malware to like a specific Facebook page, and second, to join a specific Facebook event. We use Facebook's graph search [26] to find other users that liked the specific page and accepted the specific event directed by the CnC. From this list we sampled a total of 4,596 users. Note, however, that simply because a user matched the two filters does not necessarily mean they are compromised by Febipos.A.

To improve our confidence in compromised users, we clustered the posts (based on content similarity) made to these users' walls and manually inspected the top 20 most common posts. Among these 20 posts, two posts looked suspicious. Upon further investigation, we found out that one of the post was also found on pages the malware was directed to like. The other post was present in the CnC logs we collected. The first was posted by 1,173 users while the second was posted by 135 users. We considered users from both these clusters and obtained a set of 1,193 unique users.[6] We collected 247,589 likes and 916,613 Timeline posts from their profile.

6.2 Ethics

We note that all money we paid to acquire anomalous likes were exclusively for pages both controlled by us and setup for the sole purpose of conducting the experiments in this paper. For the malware analysis, we ensured that our sandbox prevented the malware from executing the CnC's instructions. We did not seek or receive any account credentials of any Facebook user. Overall, we ensured that no other Facebook page or user was harmed or benefited as a result of this research experiment.

6.3 Normal User Ground Truth

We collected three datasets to capture normal user behavior. The first dataset is the 719 users that are part of the SIGCOMM [33] and COSN [24] Facebook groups. We picked these technically savvy users, despite the obvious bias, because we presume that these users are less likely to be infected by browser or other malware which we have found to be stealthy enough to avoid detection by non-technically-savvy users. An anomaly detector that has low false-positives on both this dataset as well as a more representative Facebook dataset is more likely to have a range that spans the spectrum of user behavior on Facebook.

[6]The friendship network formed by these users has a very low edge density of 0.00023. Thus, even though they had similar posts on their Timeline, very few of them were friends with each other (further suggesting suspicious behavior).

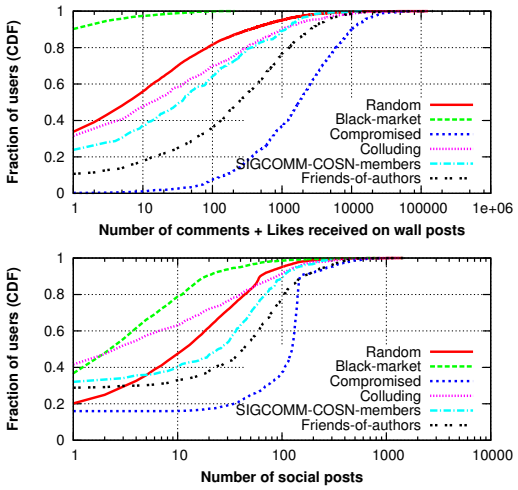


Figure 2: **Characterizing social activity of normal and anomalous users considered in our study based on activity on their Timeline.**

For our second dataset, we use the random sampling of Facebook users described in Section 4.1. Note that this dataset may be biased in the opposite direction: while it is representative of Facebook users, an unknown fraction of them are fake, compromised, or colluding. Public estimates lower-bound the number of fake users at 3% [32], thus we expect some anomalies in this dataset.

A compromise between the two extremes is our third dataset: a 1-hop crawl of the social-neighborhood of the authors (a total of 1,889 users). This dataset is somewhat more representative of Facebook than the first dataset, and somewhat less likely to be fake, compromised, or colluding than the second dataset. Users in these three datasets in total had 932,704 likes and 2,456,864 Timeline posts putting their level of activity somewhere between the black-market service on the low end, and compromised users on the high end. This fact demonstrates the challenges facing anomaly detectors based on simplistic activity thresholds.

For the rest of the analysis in this paper, we use the random sampling dataset for training our anomaly detector, and the other two datasets for testing normal users.

Figure 2 plots the cumulative distribution (CDF) of likes and comments received on wall posts and the number of *social*[7] posts for all of our six datasets. The top figure plots the CDF of likes and comments on a logarithmic x-axis ranging from 1 to 1M, and the bottom figure plots the CDF of social posts (messages, URLs, photos). As is evident from the figure, black-market users are the least active, compromised users are the most active, and all three normal user datasets—as well as the collusion network users—fall in the middle and are hard to distin-

[7]Posts that involve interaction with other users, e.g., photo tagging.

	Random	Normal	Black-market	Compromised	Colluding
#Users (#likes)	11,851 (561,559)	1,274 (73,388)	3,254 (1,544,107)	1,040 (209,591)	902 (277,600)

Table 1: **Statistics of different types of users whose like activity (from June 2011 to August 2013) we analyze.**

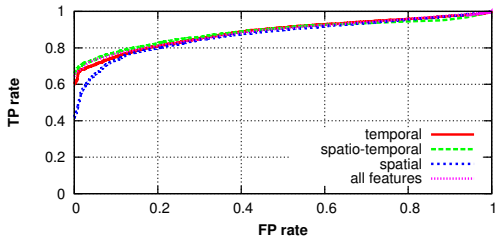


Figure 3: **ROC curve showing the performance of our anomaly detector in distinguishing between normal and misbehaving users.**

guish visually (especially for social post activity).

6.4 Detection Accuracy

Methodology: We analyze Facebook like activity from June 2011 to August 2013. We need to pay special attention to users that joined Facebook in the middle of our analysis period (or stopped being active) to avoid the degenerate case where the anomaly detection flags their lack of activity. We avoid this by considering a six-month sliding window that advances by one month. In each window, we consider users that joined before that window and had at least one like during the window. Unless otherwise mentioned, for the rest of the analysis in the paper, we consider only these users and their likes that fall within our period of analysis—data statistics are shown in Table 1. A user's behavior is flagged as anomalous if they are flagged in any one of the sliding time windows. They are flagged as anomalous in a window if the squared prediction error (SPE) exceeds the threshold parameter.

We set the detection threshold (conservatively) based on Facebook's estimate (from their SEC filings [32]) of users that violate terms of service. Facebook estimates around 3.3% users in 2013 to be undesirable (spam or duplicates). Recall that we train our anomaly detector on the like behavior of random Facebook users during much of the same period. We conservatively pick a training threshold that flags 3% of random accounts, and adjust our false-positive rate downwards by the same amount and further normalize it to lie in the range 0 to 1. We select the top-five components from our PCA output to build the normal subspace.

Results: Figure 3 plots the *receiver operating characteristic* (ROC) curve of our detector when evaluated on all datasets for normal and anomalous user behavior (except random, which was used to train the detector) as we perform a parameter-sweep on the detection threshold. The y-axis plots the true-positive rate ($\frac{TP}{TP+FN}$) and the x-axis plots the false-positive rate ($\frac{FP}{FP+TN}$) where TP, TN, FP, FN are true-positive, true-negative, false-positive, and false-negative, respectively. The area under the ROC curve for an ideal classifier is 1, and that for a random classifier is 0.5. For the mix of misbehaviors represented in our ground-truth dataset, the spatio-temporal features performs best, with an area under the curve of 0.887, followed closely by temporal and spatial features at 0.885 and 0.870, respectively.

By combining the set of users flagged by all three features, our detector is able to flag 66% of all misbehaving users at a false-positive rate of 0.3%. If we compare this with a naïve approach of flagging users based on a simple like volume/day (or like categories/day) cut-off (i.e., by flagging users who exceed a certain number of likes per day or topic categories per day) we can only detect 26% (or 49%) of all misbehaving users at the same false-positive rate. This further suggests that our PCA-based approach is more effective than such naïve approaches at capturing complex normal user behavior patterns to correctly flag misbehaving users.

Figure 4 and Table 2 explore how the set of features performed on the three classes of anomalous behavior. Spatio-temporal features alone flagged 98% of all activity for users acquired through the four black-market services. 61% (939K) of black-market activity was flagged as anomalous by all three sets of features. Due to the dominant nature of the spatio-temporal features on the black-market dataset, there is insufficient data outside the spatio-temporal circle to draw inferences about the other features. The three features performed more evenly on the dataset of compromised and colluding users, with 43.9% and 78.7% of the anomalous user behavior respectively being flagged by all three sets of features, and 64% and 91% respectively being flagged by at least one. Except in the black-market case, no class of features dominates, and combined they flag 94.3% of all anomalous user behavior in our dataset.

6.5 Error Analysis

To better understand our false-negative rate, Figure 5 plots the likelihood of detection as a function of the level of activity (number of likes) for each class of anomalous traffic. Unlike black-market users that are easily detected at any level of activity, the anomaly detector does not flag compromised and colluding users with low activity. This

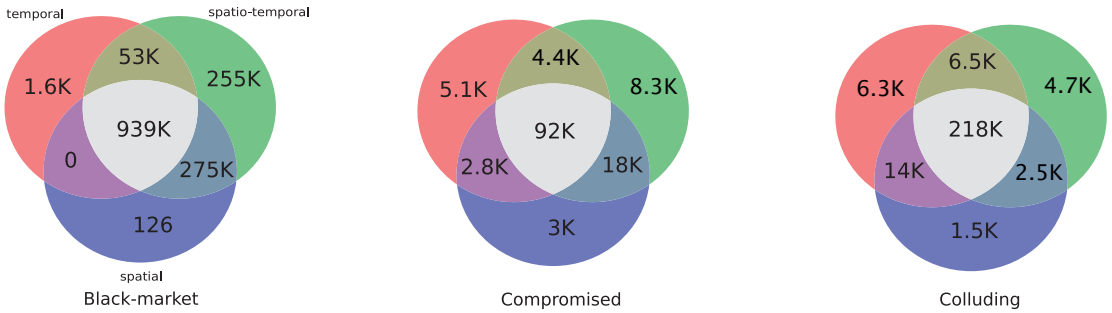


Figure 4: **Venn diagram illustrating performance of different features in detecting different classes of anomalous user behavior. The numbers indicate number of likes flagged.**

Identity type	Identities flagged	Likes flagged			
		Total	Temporal	Spatio-temporal	Spatial
Black-market	2,987/3,254 (91%)	1,526,334/1,544,107 (98%)	994,608 (64%)	1,524,576 (98%)	1,215,396 (78%)
Compromised	171/1,040 (16%)	134,320/209,591 (64%)	104,596 (49%)	123,329 (58%)	116,311 (55%)
Colluding	269/902 (29%)	254,949/277,600 (91%)	246,016 (88%)	232,515 (83%)	237,245 (85%)

Table 2: **Performance of different features in detecting different classes of anomalous user behavior.**

is consistent with compromised and colluding user behavior being a blend of normal user behavior intermixed with attacker behavior. At low levels of activity, the detector lacks data to separate anomalous behavior from noise. However, as the attacker leverages the account for more attacks, the probability of detection increases. It increases faster for colluding users, where the user is choosing to engage in anomalous activity, and more slowly for compromised accounts where the user contributes normal behavior to the blend.

Figure 6 compares anomalous user behavior that was not flagged by our detector to the behavior of normal users. As is evident from the figure, the false-negatives for compromised and colluding users appear indistinguishable from normal user behavior, especially when compared to the behavior of colluding and compromised users that were flagged. Our hypothesis (consistent with the previous paragraph) is that these false-negative users are newly compromised users or users newly recruited to the collusion network, and their overall behavior has not yet diverged significantly enough to be considered an anomaly.

Regarding false-positives, we expect some fraction of users to be flagged, since an unknown fraction of the normal users may be infected by malware. Our false-

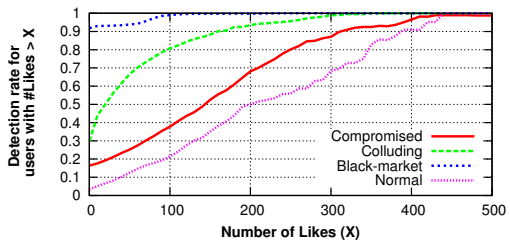


Figure 5: **Higher like activity generally correlates with higher detection rates, however limits for normal user behavior being flagged are 50–100 likes higher than for anomalous user behavior.**

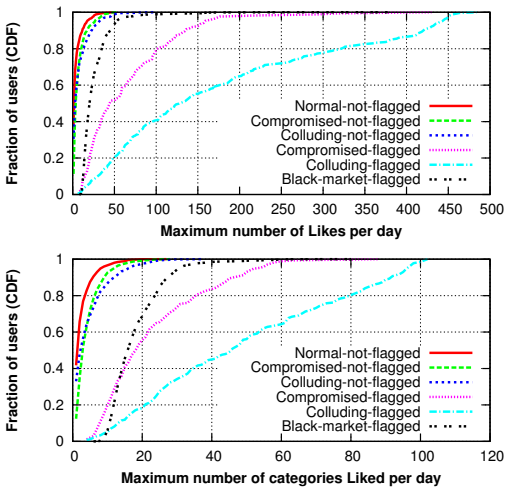


Figure 6: **Characterizing activity of users that are not flagged in the compromised and colluding set and comparing them with normal users who were not flagged.**

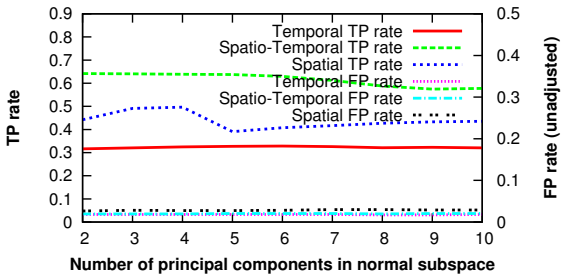


Figure 7: **False-positive rate (unadjusted) and true-positive rate as we vary the number of principal components chosen for the normal subspace. Our detector is stable for small variations in the number of principal components chosen.**

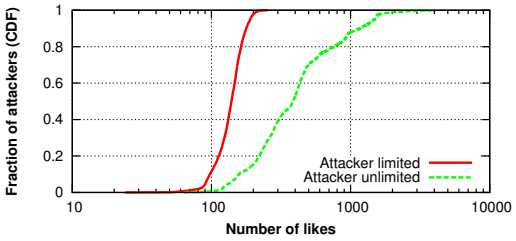


Figure 8: **Distribution of number of anomalous likes before anomalous users are flagged by our approach. For comparison, we show the actual number of anomalous likes we received.**

positive rate is under 3.3%, which when adjusted for the fraction of users Facebook expects to be anomalous [32], suggests a false-positive rate of 0.3%. We specifically note in Figure 5 that the threshold before normal user behavior is flagged is consistently 50–100 likes higher than that for compromised users for the same y-axis value. Thus, our anomaly detection technique accommodates normal users that are naturally prone to clicking on many likes.

6.6 Robustness

Next we evaluate the sensitivity of our detector to small variations in the number of principal components chosen for the normal subspace. Figure 7 plots the true-positive rate and the false-positive rate (unadjusted) as we vary k, the number of principal components used to construct the normal subspace. As is evident from the figure, our detection accuracy does not change appreciably for different choices of k. Thus our detector is quite robust to the number of principal components chosen.

6.7 Adversarial Analysis

In this section, we consider two classes of attackers: first, where the attacker scales back the attack to avoid detection, and second, where the attacker attempts to compromise the training phase.

Scaling Back: Figure 8 explores the scenario where attackers scale back their attacks to avoid detection. Specifically, we simulate the scenario where we subsample likes uniformly at random from our ground-truth attack traffic (black-market, compromised and colluding) until the point a misbehaving user is no longer flagged by the anomaly detector. As users' behavior spans multiple six month time windows, for each user we consider the window in which the user displayed maximum misbehavior (maximum number of likes in this case). In this way, we analyze the extent to which we can constrain attackers during their peak activity period. We find that our current model parameters constrains attackers by a factor of 3 in the median case, and by an order of magnitude at the 95^{th} percentile.

Compromising Training: An attacker that controls a sufficiently large number of users may attempt to compromise the training phase by injecting additional likes, thereby distorting the principal components learned for normal users [39, 55, 56]. The compromised detector would have a higher false-negative rate, since more anomalous behavior would fall within the normal subspace. At a high level, this attack may be mitigated by defense-in-depth, where multiple techniques can be used to filter users selected for the training set.

The first defense-in-depth technique is the attacker's need to control a sufficiently large number of anomalous users. We first note that our training data already contains an estimated 3% anomalous users, and that the trained detector has good performance on the ROC curve. Since users in the training set are sampled uniformly at random from all users, an attacker with equivalent power would need to be in control of over $30M$ users (given Facebook's user base of over $1B$ users). In comparison, one of the largest botnets today is estimated to have fewer than 1 million bots [47]. A related issue is that the quantity of like volume that must be injected to affect the detector depends on the overall volume of likes in the system, which is information that is not likely to be readily available to the attacker.

Assuming the attacker is able to amass this large a number of users, the next defense-in-depth technique is to sanitize training data, where anomalous users discovered in one time window are excluded from being used for training in all subsequent time windows [39]. Thus if an attacker ends up altering like traffic significantly in one time window, it could lead to detection and further

removal of those anomalous users from the training set.

Finally, variants of PCA that are more robust to outliers can be used to further harden the training phase from compromise. Croux et al. [9, 39] proposed the robust PCA-GRID algorithm that reduces the effect of outliers in the training data. Using this approach one can compute principal components that maximize a more robust measure of data dispersion – the *median absolute deviation* without under-estimating the underlying variance in the data. Such an algorithm could yield robust estimates for the normal subspace.

6.8 Scalability

As discussed earlier, classifying users can be trivially parallelized once the training phase is complete. Thus our primary focus in this section is on evaluating the scalability of the training phase.

Space: The total space requirement of the training phase is $O(n \times m)$ where n is the number of input dimensions (typically a few hundred), and m is the number of users in the training set (typically a few million). Thus the space needed to store the matrix is at most a few gigabytes, which can easily fit in a typical server's memory.

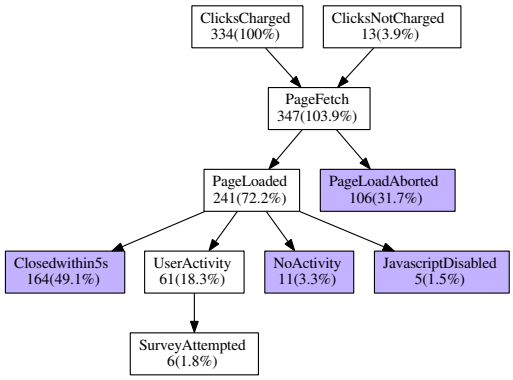
Computation: The primary computation cost in PCA arises from the eigenvalue decomposition of the covariance matrix of the feature vectors, which is a low-order polynomial time algorithm with complexity $O(n^3 + n^2 m)$. Eigenvalue decomposition is at the heart of the PageRank algorithm (used in early search engines) for which efficient systems exist to handle input data several orders of magnitude larger than our need [1]. Furthermore, efficient algorithms for PCA based on approximation and matrix sketching have been designed which have close to $O(mn)$ complexity [46, 57].

7 Detecting Click-Spam on Facebook Ads

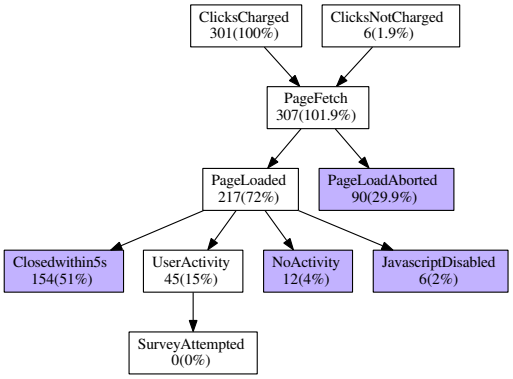
So far, we have discussed the performance of our anomaly detector in detecting diverse attack strategies. Next, we demonstrate another real world application of our technique: detecting click-spam on Facebook ads. Click-spam in online ads—where the advertiser is charged for a click that the user did not intend to make (e.g., accidental clicks, clicks by bots or malware)—is a well-known problem in web search [11, 12], and an emerging problem for Facebook ads [2, 16, 17].

7.1 Click-Spam in Facebook

To gain a preliminary understanding of Facebook click-spam, we signed up as an advertiser on Facebook. We set up an ad campaign targeting users in the USA aged between 15 and 30. The campaign advertised a simple user survey page about Facebook's privacy settings. When clicked, the ad leads to our heavily instrumented landing page to capture any user activity such as mouse clicks, mouse movement, or keyboard strokes. Of the 334 original ad clicks Facebook charged us for, only 61 (18.3%) performed any activity on the landing page (e.g., mouse move). Figure 9(a) shows how users proceeded after clicking the ad. Percentages are relative to the number of ad clicks Facebook charged us for. Shaded boxes are undesirable terminal states that suggest click-spam. For instance, 106 users (31.7%) did not even complete the first HTTP transaction to load the page (e.g., closed the tab, or pressed the back button immediately after clicking the ad).



(a) **Original ad**



(b) **Bluff ad**

Figure 9: **Summary of click statistics for real and bluff ad campaigns on Facebook.**

To distinguish between unintentional clicks and intentional clicks followed by lack of interest in our page, we ran Bluff Ads [11,38] that are ads with identical targeting parameters as the original ad, but nonsensical content. Our bluff ad content was empty. Figure 9(b) shows that our bluff ad performed identically to the original ad, both qualitatively and quantitatively; of 301 clicks in roughly the same time-frame as the original ad, almost 30% did not complete first HTTP, etc. From our data it appears

that the content of the ad has no effect on clicks on Facebook ads that we were charged for, a strong indicator of click-spam.

7.2 Anomalous Clicks in Facebook Ads

In order to analyze anomalous user behavior, our approach requires information from the user's profile. Due to a change in how Facebook redirects users on ad clicks [42], we were unable to identify the users that clicked on our ad in the experiment above. Fortunately, Facebook offers a different type of ad campaign optimization scheme—maximizing likes—where the destination must be a Facebook page as opposed to an arbitrary website. With such ads, it is possible to identify the users that clicked on such an ad, but not possible to instrument the landing page to get rich telemetry as above. We chose this campaign optimization option for maximizing likes to the advertised page.

We set up 10 ad campaigns, listed in Table 3, targeting the 18+ demographic in 7 countries: USA, UK, Australia, Egypt, Philippines, Malaysia and India. Our 10 campaigns were about generic topics such as humor, dogs, trees, and privacy awareness. Our ad contained a like button, a link to the Facebook page, some text, and an image describing the topic of the ad. We ran these ads at different points in time: Campaigns 1 to 4 were run in February 2014, while campaigns 5 to 10 were run in January 2013. In total, we received 3,766 likes for all our pages. For most of the campaigns targeting India (especially #7), we received 80% of the likes within 10 minutes, which is very anomalous.

We first checked whether we obtained most of these likes via social cascades (i.e., a user liking a page because their friend liked it), or from the Facebook ads directly. To do so, we analyzed the edge density of all friendship networks (graph formed by friendship links between users) formed by users of each ad campaign. We find the edge density of friendship networks for all campaigns to be very low (e.g., the friendship network edge density for users in campaign #8 was only 0.000032). This strongly suggests that the Facebook ads, rather than any social cascades, were responsible for the likes.

Out of 3,766 likes, we were able to crawl the identity of the users clicking like for 3,517 likes.[8] Next, we apply our anomaly detection technique from Section 5 with the same training data and model parameters that we used in Section 6 to 2,767 users (out of 3,517) who fall within our 26-month training window. The penultimate column in Table 3 lists the number of users tested in each campaign, and the last column lists the number of users flagged as click-spam.

[8]The Facebook user interface does not always show the identity of all users who like a page.

Of the 2,767 users that clicked our ads in this experiment, 1,867 were flagged as anomalous. Figure 10 plots the like activity of the users we flagged as anomalous relative to our normal user behavior dataset, and the black-market user dataset that serves as our ground-truth for anomalous user activity. The flagged users from our ad dataset have an order of magnitude more like activity than the black-market users, and nearly two orders of magnitude more like activity than normal users; they also like twice as many categories as black-market users and almost an order of magnitude more categories than normal users.

7.3 Anomaly Classification

To better understand the click-spam we observed, we attempt to classify the ad users as one of our three ground-truth anomalous behaviors: black-market, compromised, and collusion. Note that anomaly classification in this section is unrelated to the anomaly detection approach from Section 5.

We use the k-Nearest Neighbor (kNN) algorithm for classification. We train the classifier using ground-truth labels for black-market, compromised, and colluding users. The input feature vectors can be formed in different ways: First, we can capture user behavior by projecting it on to the normal and residual subspace. The normal projection reflects normal behavior and the residual projection captures noisy or deviant behavior of a user. Second, we know that user behavior can also be expressed using temporal, spatio-temporal and spatial features. By leveraging all these different combinations, we built 6 classifiers using 6 different feature vectors (2 projections

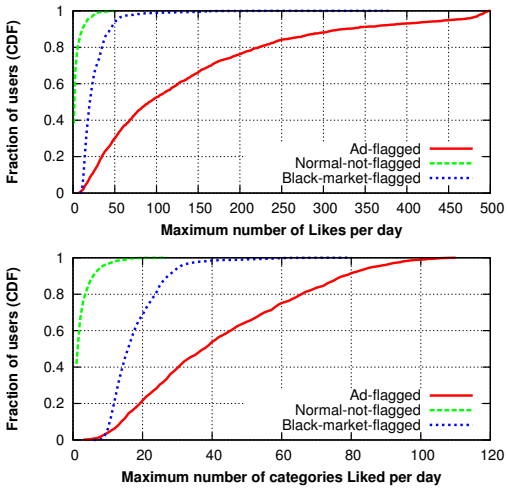


Figure 10: **Characterizing activity of users flagged in the ad set. Note that most flagged ad users like a much larger number of categories/likes per day than normal and black-market users.**

Campaign	Ad target	Cost per like (€)	Total spent (€)	Users		
				Total	Tested	Flagged
1	US	1.62	192.43	119	76	43
2	UK	1.95	230.05	118	69	27
3	AU	0.87	158.89	182	88	38
4	Egypt, Philippines, Malaysia	0.08	47.69	571	261	135
5	India	0.13	30.00	230	199	137
6	India	0.11	22.71	209	169	99
7	India	0.09	22.61	250	199	114
8	India, US, UK	0.22	242.72	1,099	899	791
9	India	0.12	30.00	247	215	143
10	India	0.07	50.00	741	632	372

Table 3: **Anomalies flagged for different ad campaigns. We observe a significant fraction of anomalous clicks for all campaigns.**

$\times$ 3 features). Each classifier, given an unlabeled user from the ad set, predicts a label for the user.

We use a simple ensemble learning technique of *majority voting* to combine the results of all the classifiers; this also means that there could be test instances that may not be labeled due to lack of consensus. We choose the most recent six-month time window (March to August 2013) in our dataset and use all known misbehaving users (black-market, compromised and colluding) in that window for training the classifier and apply this technique to the 1,408 flagged ad users who fall in that window. To balance classes for training, we randomly under-sample larger classes (black-market and colluding) and use 780 users in each of black-market, colluding and compromised set for training. For each classifier, we pick a value of k that gives the lowest misclassification rate for 10-fold cross validation on the training data. We next apply our trained classifier to predict the unlabeled ad users. Results are averaged over 50 different random trials and we observe an average misclassification rate of 31% (standard deviation of 0.5) based on cross-validation in the training phase. Table 4 shows the statistics for the labels predicted for the flagged ad users. We find that the majority of ad users (where we had majority agreement) are classified as black-market or compromised.

Classified As	Number of users
Black-market	470
Compromised	109
Colluding	345
Unclassified (no consensus)	484

Table 4: **Anomaly class predicted for the ad users that are flagged.**

While the level of anomalous click traffic is very surprising, it is still unclear what the incentives are for the attacker. One possibility is that black-market accounts and compromised accounts are clicking (liking) ads to generate cover traffic for their misbehavior. Another possibility is that the attacker is trying to drain the budget of some advertiser by clicking on ads of that advertiser. We plan to explore this further as part of future work.

8 Corroboration by Facebook

We disclosed our findings to Facebook in March 2014, and included a preprint of this paper. Our primary intent in doing so was to follow responsible disclosure procedures, and to allow Facebook to identify any ethical or technical flaws in our measurement methodology. We were informed that Facebook's automated systems detect and remove fake users and fraudulent likes.

Table 5 tabulates the users (flagged by our detector) and likes that were removed between the time we conducted our experiments and June 2014. While very few users were removed by Facebook, a sizable fraction of their likes across all pages were indeed removed confirming the accuracy of our detector. To establish a baseline for the fraction of users and likes removed by Facebook's automated systems we find that from our random user dataset (Section 4) only 2.2% users, and 32% of all their likes were removed over a ten month period. For black-market, compromised, and colluding users (ground-truth anomalous user dataset from Section 6), over 50% of all their likes had been removed over 6–10 months. Over 85% of the all likes of users that clicked our ad were removed within four months. Recall that our ad was targeted to normal Facebook users and we did not use any external services to acquire ad likes; nevertheless, 1,730 of the 3,517 likes we were charged for in February 2014 had been removed by Facebook's fraudulent like detection system by June 2014, corroborating our earlier result that a large fraction of users that clicked on our ad are anomalous both by our definition as well as Facebook's.[9] As of this writing we have not received any

[9] While Facebook allows users to un-like pages, according to Facebook insights [30] we had only 56 un-likes across all our pages, which we exclude from our analysis.

	Removed by Facebook's automated systems			
	Users	**likes on all pages**	**likes on our page**	**Timespan**
Normal User Dataset (Section 4)				
Random users	262/12K	179K/561K	n/a	10 months
Ground-Truth Anomaly Dataset (Section 6)				
Black-market	228/2987	715K/1.5M	2829/3475	10 months
Compromised	3/171	80K/134K	n/a	7 months
Colluding	9/269	181K/254K	1879/2259	6 months
Facebook Ads Dataset (Section 7)				
Ad clicks	51/1867	2.9M/3.4M	1730/3517[9]	4 months

Table 5: **Fraction of users and likes flagged by us removed by Facebook's automated system, as of June 2014.**

credit adjustments for the likes charged to our advertiser account that Facebook's fraudulent like detection system since identified and removed.

9 Related Work

We survey approaches to detecting misbehaving identities along three axes.

Leveraging Hard-to-earn Attributes: Manual verification of users would be ideal to avoiding Sybils in crowdsourcing systems but does not scale for large-scale systems. Additionally, normal users may not join the system for privacy reasons due to the effort required to be verified. Current systems typically employ CAPTCHA or phone verification to raise the barrier by forcing the attacker to expend greater effort. Although pervasive, attackers try to evade these schemes by employing Sybil identities that use sites like Freelancer or Amazon's Mechanical Turk to exploit the differences in value of human time in different countries [51]. However, steps taken by service providers to raise the barrier for fake account creation complements our proposed defense because each account flagged as anomalous raises the cost for the attacker.

In OSNs, where identities are associated with each other through hard-to-earn endorsement and friend edges, several graph-based Sybil detection schemes have been developed over the years [10, 52, 61, 66, 67]. Such schemes make assumptions about the OSN graph growth and structure, for example that creating and maintaining edges to honest identities requires significant effort [48], or that honest OSN regions are fast-mixing [66, 67]. However, recent studies cast doubts on these assumptions and subsequently on the graph-based Sybil defense techniques. Specifically, Yang et al. [65] observe that Sybils blend well into the rest of OSN graphs, while Mohaisen et al. [49] find that most OSN graphs are not fast-mixing, and that detection schemes may end up accepting Sybil identities and/or wrongly expelling honest identities [62].

Supervised Learning: Most existing work on detecting misbehaving identities in social networks leverage supervised learning techniques [14,40,53]. Lee et al. [40] propose a scheme that deploys honeypots in OSNs to attract spam, trains a machine learning (ML) classifier over the captured spam, and then detects new spam using the classifier. Rahman et al. [53] propose a spam and malware detection scheme for Facebook using a Support Vector Machines-based classifier trained using the detected malicious URLs. The COMPA scheme [14] creates statistical behavioral profiles for Twitter users, trains a statistical model with a small manually labeled dataset of both benign and misbehaving users, and then uses it to detect compromised identities in Twitter.

While working with large crowdsourcing systems, supervised learning approaches have inherent limitations. Specifically they are attack-specific and vulnerable to *adaptive* attacker strategies. Given the adaptability of the attacker strategies, to maintain efficacy, supervised learning approaches require labeling, training, and classification to be done periodically. In this cat-and-mouse game, they will always lag behind attackers who keep adapting to make a classification imprecise.

Unsupervised Learning: Unsupervised learning-based anomaly detection has been found to be an effective alternative to non-adaptive supervised learning strategies [12, 45, 60, 63, 64]. For example, Li et al. [45] propose a system to detect volume anomalies in network traffic using unsupervised PCA-based methods. AutoRE [64] automatically extracts spam URL patterns in email spam based on detecting the bursty and decentralized nature of botnet traffic as anomalous.

In crowdsourcing scenarios, Wang et al. [63] proposed a Sybil detection technique using server-side clickstream models (based on user behavior defined by clickthrough events generated by users during their social network browsing sessions). While the bulk of the paper presents supervised learning schemes to differentiate between Sybil and non-Sybils based on their clickstream behavior, they also propose an unsupervised approach

that builds clickstream behavioral clusters that capture normal behavior and users that are not part of normal clusters are flagged as Sybil. However, their approach still requires some constant amount of ground-truth information to figure out clusters that represent normal click-stream behavior. Tan et al. [60] use a user-link graph along with the OSN graph to detect some honest users with supervised ML classifier and then perform an unsupervised analysis to detect OSN spam. CopyCatch [3] detects fraudulent likes by looking for a specific attack signature — groups of users liking the same page at around the same time (*lockstep behavior*). CopyCatch is actively used in Facebook to detect fraudulent likes, however as evidenced in Table 5, it is not a silver-bullet.

While we welcome the push towards focusing more on unsupervised learning strategies for misbehavior detection, most of the current techniques are quite ad hoc and complex. Our approach using Principal Component Analysis provides a more systematic and general framework for modeling user behavior in social networks, and in fact, our PCA-based approach could leverage the user behavior features (e.g., user click-stream models [63]) used in existing work for misbehavior detection.

10 Conclusion

We propose using Principal Component Analysis (PCA) to detect anomalous user behavior in online social networks. We use real data from three social networks to demonstrate that normal user behavior is low-dimensional along a set of latent features chosen by PCA. We also evaluate our anomaly detection technique using extensive ground-truth data of anomalous behavior exhibited by fake, compromised, and colluding users. Our approach achieves a detection rate of over 66% (covering more than 94% of misbehavior) with less than 0.3% false positives. Notably we need no *a priori* labeling or tuning knobs other than a configured acceptable false positive rate. Finally, we apply our anomaly detection technique to effectively identify anomalous likes on Facebook ads.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Rachel Greenstadt, for their helpful comments. We also thank Arash Molavi Kakhki for his assistance with the Yelp dataset and Mainack Mondal for helping with the ad experiments. This research was supported in part by NSF grants CNS-1054233, CNS-1319019, CNS-0905565, CNS-1018266, CNS-1012910, and CNS-1117039, and by the Army Research Office under grant W911NF-11-1-0227.

References

[1] Apache Mahout. http://mahout.apache.org/.

[2] Facebook Advertisement Value (BBC). http://www.bbc.co.uk/news/technology-18813237.

[3] BEUTEL, A., XU, W., GURUSWAMI, V., PALOW, C., AND FALOUTSOS, C. Copycatch: Stopping group attacks by spotting lockstep behavior in social networks. In *Proc. of WWW* (2013).

[4] BHATTACHARYA, P., GHOSH, S., KULSHRESTHA, J., MONDAL, M., ZAFAR, M. B., GANGULY, N., AND GUMMADI, K. P. Deep Twitter Diving: Exploring Topical Groups in Microblogs at Scale. In *Proc. of CSCW* (2014).

[5] CATTELL, R. B. The Scree Test For The Number Of Factors. *Journal of Multivariate Behavioral Research 1*, 2 (1966).

[6] Click Farms (Washington Post). http://www.washingtonpost.com/blogs/wonkblog/wp/2014/01/06/click-farms-are-the-new-sweatshops.

[7] Collusion Network Site #1. http://addmefast.com/.

[8] Collusion Network Site #2. http://likesasap.com/.

[9] CROUX, C., FILZMOSER, P., AND OLIVEIRA, M. Algorithms for ProjectionPursuit Robust Principal Component Analysis. *Journal of Chemometrics and Intelligent Laboratory Systems 87*, 2 (2007).

[10] DANEZIS, G., AND MITTAL, P. SybilInfer: Detecting Sybil Nodes Using Social Networks. In *Proc. of NDSS* (2009).

[11] DAVE, V., GUHA, S., AND ZHANG, Y. Measuring and Fingerprinting Click-Spam in Ad Networks. In *Proc. of SIGCOMM* (2012).

[12] DAVE, V., GUHA, S., AND ZHANG, Y. ViceROI: Catching Click-Spam in Search Ad Networks. In *Proc. of CCS* (2013).

[13] DING, Q., AND KOLACZYK, E. D. A Compressed PCA Subspace Method for Anomaly Detection in High-Dimensional Data. *IEEE Transactions on Information Theory 59*, 11 (2013).

[14] EGELE, M., STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. COMPA: Detecting Compromised Accounts on Social Networks. In *Proc. of NDSS* (2013).

[15] Facebook Accounts Compromised. http://mashable.com/2013/12/04/hackers-facebook-twitter-gmail/.

[16] Facebook Ads Fake Clicks. http://www.businessinsider.com/mans-600000-facebook-ad-disaster-2014-2.

[17] Facebook Advertisement Value (Salon). http://www.salon.com/2014/02/14/facebooks_big_like_problem_major_money_and_major_scams/.

[18] Facebook Black Market Service #1. http://fbviro.com.

[19] Facebook Black Market Service #2. http://get-likes.com/facebook-likes-store/.

[20] Facebook Black Market Service #3. http://www.buyfbservices.com.

[21] Facebook Black Market Service #4. http://twittertechnology.com/facebook.

[22] Facebook Black Market Service #5. http://www.pagelution.com/buy-facebook-fans.html.

[23] Facebook Black Market Service #6. http://teamfollowpromo.com/facebooklikes.

[24] Facebook COSN Group. https://www.facebook.com/groups/cosn2013.

[25] Facebook Directory. https://www.facebook.com/directory/people/.

[26] Facebook Graph Search. https://www.facebook.com/about/graphsearch.

[27] Fake Likes on Facbook and Instagram. http://www.huffingtonpost.com/2013/08/16/fake-instagram-likes_n_3769247.html.

[28] Facebook Malware Extension. http://bit.ly/1mIum7L.

[29] Facebook Internet Explorer Malware. http://blogs.technet.com/b/mmpc/archive/2013/11/14/febipos-for-internet-explorer.aspx.

[30] Facebook Page insights. https://www.facebook.com/help/336893449723054.

[31] Facebook Quarterly Report (2013). http://investor.fb.com/releasedetail.cfm?ReleaseID=821954.

[32] Facebook's estimate of fraction of undesirable accounts for 2013. http://investor.fb.com/.

[33] Facebook SIGCOMM Group. https://www.facebook.com/groups/sigcomm/.

[34] Facebook Timeline. https://www.facebook.com/about/timeline.

[35] Febipos.A Malware. http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Trojan:JS/Febipos.A.

[36] GAO, H., HU, J., WILSON, C., LI, Z., CHEN, Y., AND ZHAO, B. Y. Detecting and Characterizing Social Spam Campaigns. In *Proc. of IMC* (2010).

[37] GRIER, C., THOMAS, K., PAXSON, V., AND ZHANG, C. M. @spam: The Underground on 140 Characters or Less. In *Proc. of CCS* (2010).

[38] HADDADI, H. Fighting Online Click-Fraud Using Bluff Ads. *ACM SIGCOMM Computer Communications Review 40*, 2 (2010).

[39] HUANG, L., JOSEPH, A. D., NELSON, B., RUBINSTEIN, B. I., AND TYGAR, J. D. Adversarial Machine Learning. In *Proc. of AISec* (2011).

[40] K. LEE AND J. CAVERLEE AND AND S. WEBB. Uncovering Social Spammers: Social Honeypots + Machine Learning. In *Proc. of SIGIR* (2010).

[41] KOREN, Y. Collaborative filtering with temporal dynamics. In *Proc. of KDD* (2009).

[42] KRISHNAMURTHY, B., AND WILLS, C. E. On the Leakage of Personally Identifiable Information via Online Social Networks. *ACM SIGCOMM Computer Communications Review 40*, 1 (2010).

[43] L2 Norm. http://mathworld.wolfram.com/L2-Norm.html.

[44] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing Network-wide Traffic Anomalies. In *Proc. of SIGCOMM* (2004).

[45] LI, X., BIAN, F., CROVELLA, M., DIOT, C., GOVINDAN, R., IANNACCONE, G., AND LAKHINA, A. Detection and Identification of Network Anomalies Using Sketch Subspaces. In *Proc. of IMC* (2006).

[46] LIBERTY, E. Simple and Deterministic Matrix Sketching. In *Proc. of KDD* (2013).

[47] Zeus Botnet Cleanup (Microsoft). http://blogs.technet.com/b/microsoft_blog/archive/2014/06/02/microsoft-helps-fbi-in-gameover-zeus-botnet-cleanup.aspx.

[48] MISLOVE, A., POST, A., GUMMADI, K. P., AND DRUSCHEL, P. Ostra: Leverging Trust to Thwart Unwanted Communication. In *Proc. of NSDI* (2008).

[49] MOHAISEN, A., YUN, A., AND KIM, Y. Measuring the Mixing Time of Social Graphs. Tech. rep., University of Minnesota, 2010.

[50] MOLAVI KAKHKI, A., KLIMAN-SILVER, C., AND MISLOVE, A. Iolaus: Securing Online Content Rating Systems. In *Proc. of WWW* (2013).

[51] MOTOYAMA, M., MCCOY, D., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. Dirty Jobs: The Role of Freelance Labor in Web Service Abuse. In *Proc. of USENIX Security* (2011).

[52] QUERCIA, D., AND HAILES, S. Sybil Attacks Against Mobile Users: Friends and Foes to the Rescue. In *Proc. of INFOCOM* (2010).

[53] RAHMAN, M. S., HUANG, T.-K., MADHYASTHA, H. V., AND FALOUTSOS, M. Efficient and Scalable Socware Detection in Online Social Networks. In *Proc. of USENIX Security* (2012).

[54] RINGBERG, H., SOULE, A., REXFORD, J., AND DIOT, C. Sensitivity of pca for traffic anomaly detection. In *Proc. of SIGMETRICS* (2007).

[55] RUBINSTEIN, B. I. P., NELSON, B., HUANG, L., JOSEPH, A. D., HON LAU, S., RAO, S., TAFT, N., AND TYGAR, J. D. Stealthy Poisoning Attacks on PCA-based Anomaly Detectors. *ACM SIGMETRICS Performance Evaluation Review 37*, 2 (2009).

[56] RUBINSTEIN, B. I. P., NELSON, B., HUANG, L., JOSEPH, A. D., LAU, S.-H., TAFT, N., AND TYGAR, D. Compromising PCA-based Anomaly Detectors for Network-Wide Traffic. Tech. rep., EECS Department, University of California, Berkeley, 2008.

[57] SHARMA, A., AND PALIWAL, K. K. Fast Principal Component Analysis Using Fixed-point Algorithm. *Journal of Pattern Recognition Letters 28*, 10 (2007).

[58] STRINGHINI, G., EGELE, M., KRUEGEL, C., AND VIGNA, G. Poultry Markets: On the Underground Economy of Twitter Followers. In *Proc. of WOSN* (2012).

[59] STRINGHINI, G., WANG, G., EGELE, M., KRUEGEL, C., VIGNA, G., ZHENG, H., AND ZHAO, B. Y. Follow the Green: Growth and Dynamics in Twitter Follower Markets. In *Proc. of IMC* (2013).

[60] TAN, E., GUO, L., CHEN, S., ZHANG, X., AND ZHAO, Y. E. UNIK: Unsupervised Social Network Spam Detection. In *Proc. of CIKM* (2013).

[61] TRAN, N., LI, J., SUBRAMANIAN, L., AND CHOW, S. S. Optimal Sybil-resilient Node Admission Control. In *Proc. of INFOCOM* (2011).

[62] VISWANATH, B., POST, A., GUMMADI, K. P., AND MISLOVE, A. An Analysis of Social Network-based Sybil Defenses. In *Proc. of SIGCOMM* (2010).

[63] WANG, G., KONOLIGE, T., WILSON, C., WANG, X., ZHENG, H., AND ZHAO, B. Y. You Are How You Click: Clickstream Analysis for Sybil Detection. In *Proc. of USENIX Security* (2013).

[64] XIE, Y., YU, F., ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming Botnets: Signatures and Characteristics. In *Proc. of SIGCOMM* (2008).

[65] YANG, Z., WILSON, C., WANG, X., GAO, T., ZHAO, B. Y., AND DAI, Y. Uncovering Social Network Sybils In the Wild. In *Proc. of IMC* (2011).

[66] YU, H., GIBBONS, P. B., KAMINSKY, M., AND XIAO, F. SybilLimit: A Near-optimal Social Network Defense Against Sybil Attacks. In *Proc. of IEEE S &P* (2008).

[67] YU, H., KAMINSKY, M., GIBBONS, P. B., AND FLAXMAN, A. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *Proc. of SIGCOMM* (2006).

Man vs. Machine: Practical Adversarial Detection of Malicious Crowdsourcing Workers

Gang Wang[†], Tianyi Wang[†‡], Haitao Zheng[†] and Ben Y. Zhao[†]
[†]*Computer Science, UC Santa Barbara* [‡]*Electronic Engineering, Tsinghua University*
{gangw, tianyi, htzheng, ravenben}@cs.ucsb.edu

Abstract

Recent work in security and systems has embraced the use of machine learning (ML) techniques for identifying misbehavior, *e.g.* email spam and fake (Sybil) users in social networks. However, ML models are typically derived from *fixed* datasets, and must be periodically retrained. In adversarial environments, attackers can adapt by modifying their behavior or even sabotaging ML models by polluting training data.

In this paper[1], we perform an empirical study of adversarial attacks against machine learning models in the context of detecting malicious crowdsourcing systems, where sites connect paying users with workers willing to carry out malicious campaigns. By using human workers, these systems can easily circumvent deployed security mechanisms, *e.g.* CAPTCHAs. We collect a dataset of malicious workers actively performing tasks on Weibo, China's Twitter, and use it to develop ML-based detectors. We show that traditional ML techniques are accurate (95%–99%) in detection but can be highly vulnerable to adversarial attacks, including simple *evasion attacks* (workers modify their behavior) and powerful *poisoning attacks* (where administrators tamper with the training set). We quantify the robustness of ML classifiers by evaluating them in a range of practical adversarial models using ground truth data. Our analysis provides a detailed look at practical adversarial attacks on ML models, and helps defenders make informed decisions in the design and configuration of ML detectors.

1 Introduction

Today's computing networks and services are extremely complex systems with unpredictable interactions between numerous moving parts. In the absence of accurate deterministic models, applying Machine Learning (ML) techniques such as decision trees and support vector machines (SVMs) produces practical solutions to a variety of problems. In the security context, ML techniques can extract statistical models from large noisy datasets, which have proven accurate in detecting misbehavior and attacks, *e.g.* email spam [35, 36], network intrusion attacks [22, 54], and Internet worms [29]. More recently, researchers have used them to model and detect malicious users in online services, *e.g.* Sybils in social networks [42, 52], scammers in e-commerce sites [53] and fraudulent reviewers on online review sites [31].

Despite a wide range of successful applications, machine learning systems have a weakness: they are vulnerable to adversarial countermeasures by attackers aware of their use. First, through either reading publications or self-experimentation, attackers may become aware of details of the ML detector, *e.g.* choice of classifier and parameters used, and modify their behavior to *evade* detection. Second, more powerful attackers can actively tamper with the ML models by polluting the training set, reducing or eliminating its efficacy. Adversarial machine learning has been studied by prior work from a theoretical perspective [6, 12, 27], using simplistic all-or-nothing assumptions about adversaries' knowledge about the ML system in use. In reality, however, attackers are likely to gain incomplete information or have partial control over the system. An accurate assessment of the robustness of ML techniques requires evaluation under *realistic* threat models.

In this work, we study the robustness of machine learning models against practical adversarial attacks, in the context of detecting malicious crowdsourcing activity. Malicious crowdsourcing, also called crowdturfing, occurs when an attacker pays a group of Internet users to carry out malicious campaigns. Recent crowdturfing attacks ranged from "artificial grassroots" political campaigns [32, 38], product promotions that spread false rumors [10], to spam dissemination [13, 39]. Today, these campaigns are growing in popularity in dedicated

[1]Our work received approval from our local IRB review board.

crowdturfing sites, *e.g.* ZhuBaJie (ZBJ)[2] and SanDaHa (SDH)[3], and generic crowdsourcing sites [26, 48].

The detection of crowdturfing activity is an ideal context to study the impact of adversarial attacks on machine learning tools. First, crowdturfing is a growing threat to today's online services. Because tasks are performed by intelligent individuals, these attacks are undetectable by normal measures such as CAPTCHAs or rate limits. The results of these tasks, fake blogs, slanderous reviews, fake social network accounts, are often indistinguishable from the real thing. Second, centralized crowdturfing sites like ZBJ and SDH profit directly from malicious crowdsourcing campaigns, and therefore have strong monetary incentive and the capability to launch adversarial attacks. These sites have the capability to modify aggregate behavior of their users through interface changes or explicit policies, thereby either helping attackers evade detection or polluting data used as training input to ML models.

Datasets. For our analysis, we focus on Sina Weibo, China's microblogging network with more than 500 million users, and a frequent target of crowdturfing campaigns. Most campaigns involve paying users to retweet spam messages or to follow a specific Weibo account. We extract records of 20,416 crowdturfing campaigns (1,012,923 tasks) published on confirmed crowdturfing sites over the last 3 years. We then extract a 28,947 Weibo accounts belonging to crowdturfing workers. We analyze distinguishing features of these accounts, and build detectors using multiple ML models, including SVMs, Bayesian, Decision Trees and Random Forests.

We seek answers to several key questions. First, can machine learning models detect crowdturfing activity? Second, once detectors are active, what are possible countermeasures available to attackers? Third, can adversaries successfully manipulate ML models by tampering with training data, and if so, can such efforts succeed in practice, and which models are most vulnerable?

Adversarial Attack Models. We consider two types of practical adversarial models against ML systems: those launched by individual *crowd-workers*, and those involving coordinated behavior driven by administrators of centralized *crowdturfing sites*. First, individual workers can perform *evasion* attacks, by adapting behavior based on their knowledge of the target classifier (*e.g.* ML algorithms, feature space, trained models). We identify a range of threat models that vary the amount of knowledge by the adversary. The results should provide a comprehensive view of how vulnerable ML systems to evasion, ranging from the worst case (total knowledge by attacker) to more practically scenarios. Second, more powerful attacks are possible with the help of crowdturfing site administrators, who can manipulate ML detectors by *poisoning* or polluting training data. We study the impact on different ML algorithms from two pollution attacks: injecting false data samples, and altering existing data samples.

Our study makes four key contributions:

- We demonstrate the efficacy of ML models for detecting crowdturfing activity. We find that Random Forests perform best out of multiple classifiers, with 95% detection accuracy overall and 99% for "professional" workers.
- We develop adversarial models for *evasion attacks* ranging from optimal evasion to more practical/limited strategies. We find while such attacks can be very powerful in the optimal scenario (attacker has total knowledge), practical attacks are significantly less effective.
- We evaluate a powerful class of *poison* attacks on ML training data and find that *injecting* carefully crafted data into training data can significantly reduce detection efficacy.
- We observe a consistent tradeoff between fitting accuracy and robustness to adversarial attacks. More accurate fits (especially to smaller, homogeneous populations) make models more vulnerable to deviations introduced by adversaries. The exception is Random Forests, which naturally supports fitting to multiple populations, thus allowing it to maintain both accuracy and robustness in our tests.

To the best of our knowledge, this is the first study to examine automated detection of large-scale crowdturfing activity, and the first to evaluate adversarial attacks against machine learning models in this context. Our results show that accurate models are often vulnerable to adversarial attacks, and that robustness against attacks should be a primary concern when selecting ML models.

2 Datasets and Methodology

In this section, we provide background on crowdturfing, and introduce our datasets and methodology.

2.1 Background: Crowdturfing Systems

Malicious crowdsourcing (crowdturfing) sites are web services where attackers pay groups of human workers to perform questionable (and often malicious) tasks. While these services are growing rapidly world-wide, two of the largest are Chinese sites ZhuBaJie (ZBJ) and SanDaHa (SDH) [48]. Both sites leave records of campaigns publicly visible to recruit new workers, making it possible for us to crawl their data for analysis.

[2] http://www.zhubajie.com/c-tuiguang/
[3] http://www.sandaha.com/

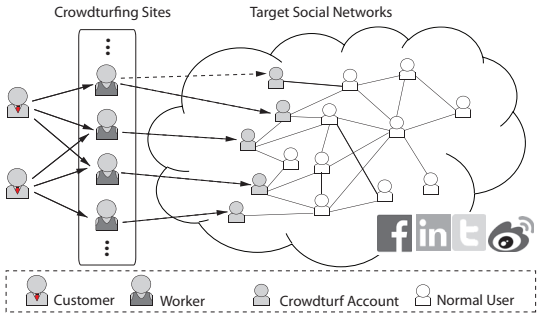


Figure 1: Crowdturfing process.

Figure 1 illustrates how crowdturfing campaigns work. Initially, a *customer* posts a campaign onto the crowdturfing site, and pays the site to carry it out. Each campaign is a collection of small *tasks*, *e.g.* tasks to send or retweet messages advertising a malware site. *Workers* accept the task, and use their fake accounts in the target social network(s) (*e.g.* Twitter) to carry out the tasks. Today, crowdturfing campaigns often spam web services such as social networks, online review sites, and instant-messaging networks [48]. While workers can be any Internet user willing to spam for profit, customers often require workers to use "high quality" accounts (*i.e.* established accounts with real friends) to perform tasks [48]. In the rest of the paper, we refer workers' social network accounts as *crowdturf accounts*.

Crowdturfing on Weibo. Sina Weibo is China's most popular microblogging social network with over 500 million users [30]. Like Twitter, Weibo users post 140-character *tweets*, which can be *retweeted* by other users. Users can also *follow* each other to form asymmetric social relationships. Unlike Twitter, Weibo allows users to have conversations via *comments* on a tweet.

Given its large user population, Weibo is a popular target for crowdturfing systems. There are two major types of crowdturfing campaigns. One type asks workers to follow a customer's Weibo account to boost their perceived popularity and visibility in Weibo's ranked social search. A second type pays crowd-workers to retweet spam messages or URLs to reach a large audience. Both types of campaigns directly violate Weibo's ToS [2]. A recent statement (April 2014) from a Weibo administrator shows that Weibo has already begun to take action against crowdturfing spam [1].

2.2 Ground Truth and Baseline Datasets

Our study utilizes a large *ground-truth* dataset of crowdturfing worker accounts. We extract these accounts by filtering through records of all campaigns and tasks targeting Weibo from ZBJ and SDH, and extracting all Weibo accounts that accepted these tasks. This is possible because ZBJ and SDH keep complete records of campaigns and transaction details (*i.e.* workers who completed tasks, and their Weibo identities) visible.

Category	# Weibo IDs	# (Re) Tweets	# Comments
Turfing	28,947	18,473,903	15,970,215
Authent.	71,890	7,600,715	13,985,118
Active	371,588	34,164,885	75,335,276

Table 1: Dataset summary.

As of March 2013, we collected a total of 20,416 Weibo campaigns (over 3 years for ZBJ and SDH), with a total of 1,012,923 individual tasks. We extracted 34,505 unique Weibo account IDs from these records. 5,558 of which have already been blocked by Weibo. We collected user profiles for the remaining 28,947 active accounts, including social relationships and the latest 2000 tweets from each account. These accounts have performed at least one crowdturfing task. We refer to this as the *Turfing* dataset.

Baseline Datasets for Comparison. We need a baseline dataset of "normal" users for comparison. We start by snowball sampling a large collection of Weibo accounts[4]. We ran breadth-first search (BFS) in November 2012 using 100 Seeds randomly chosen from Weibo's public tweet stream, giving us 723K accounts. Because these crawled accounts can include malicious accounts, we need to do further filtering to obtain a real set of "normal" users.

We extract two different baseline datasets. First, we construct a conservative *Authenticated* dataset, by including only Weibo users who have undergone an optional identity verification by phone number or Chinese national ID (equivalent to US drivers license). A user who has bound her Weibo account to her real-world identity can be held legally liable for her actions, making these authenticated accounts highly unlikely to be used as crowdturfing activity. Our *Authenticated* dataset includes 71,890 accounts from our snowball sample. Second, we construct a larger, more inclusive baseline set of *Active* users. We define this set as users with at least 50 followers and 10 tweets (filtering out dormant accounts[5] and Sybil accounts with no followers). We also cross reference these users against all known crowdturfing sites to remove any worker accounts. The resulting dataset includes 371,588 accounts. While it is not guaranteed to be 100% legitimate users, it provides a broader user sample that is more representative of average user behavior.

[4]Snowball crawls start from an initial set of seed nodes, and runs breadth-first search to find all reachable nodes in the social graph [3].

[5]Dormant accounts are unlikely to be workers. To qualify for jobs, ZBJ/SDH workers must meet minimum number of followers/tweets.

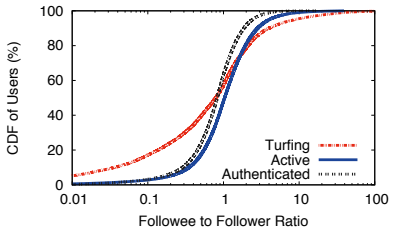


Figure 2: Followee-to-Follower ratio.

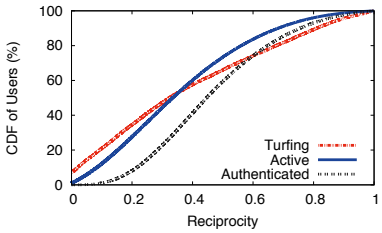


Figure 3: Reciprocity.

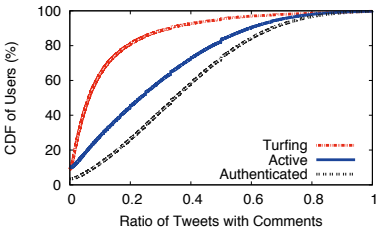


Figure 4: Ratio of commented tweets.

This is likely to provide a lower bound for detector accuracy, since more carefully curated baselines would produce higher detection accuracy. Our datasets are listed in Table 1.

2.3 Our Methodology

We have two goals: evaluating the efficacy of ML classifiers to detect crowdturfing workers, and evaluating the practical impact of adversarial attacks on ML classifiers.

- We analyze ground-truth data to identify key behavioral features that distinguish crowdturfing worker accounts from normal users (§3).
- We use these features to build a number of popular ML models, including Bayesian probabilistic models via Bayes' theorem (*i.e.* conditional probability), Support Vector Machines (SVMs), and algorithms based on single or multiple decision trees (*e.g.* Decision Trees, Random Forests) (§4).
- We evaluate ML models against adversarial attacks ranging from weak to strong based on level of knowledge by attackers (typically evasion attacks), and coordinated attacks potentially guided by centralized administrators (possible poison or pollution of training data).

3 Profiling Crowdturf Workers

We begin our study by searching for behavioral features that distinguish worker accounts from normal users. These features will be used to build ML detectors in §4.

User Profile Fields. We start with user profile features commonly used as indicators of abnormal behavior. These features include followee-to-follower ratio (FFRatio), reciprocity (*i.e.* portion of user's followees who follow back), user tweets per day, account age, and ratio of tweets with URLs and mentions.

Unfortunately, our data shows that most of these features alone cannot effectively distinguish worker accounts from normal users. First, FFRatio and reciprocity are commonly used to identify malicious spammers [4, 43, 50]. Intuitively, spammers follow a large number of random users and hope for them to follow back, thus they have high FFRatio and low reciprocity. However, our analysis shows worker accounts have balanced FFRatios, the majority of them even have more followers than followees (Figure 2), and their reciprocity is very close to those of normal users (Figure 3). Other profile features are also ineffective, including account age, tweets per day, ratio of tweets with URLs and mentions. For example, existing detectors usually assume attackers create many "fresh" accounts to spam [4, 43], thus account age has potential. But we find that more than 75% of worker accounts in our dataset have been active for at least one year.

These results show that crowd-worker accounts in many respects resemble normal users, and are not easily detected by profile features alone [47].

User Interactions. Next, we move on to features related to user interactions. The intuition is that crowdturf workers are task-driven, and log on to work on tasks, but spend minimal time interacting with others. User interactions in Weibo are dominated by comments and retweets. We perform analysis on both of them and find consistent results which show they are good metrics to distinguish workers from non-workers. For brevity, we limit our discussion to results on comment interactions.

Figure 4 shows crowdturf accounts are less likely to receive comments on their tweets. For 80% of crowdturf accounts, less than 20% of their tweets are commented; while for 70% of normal users, their ratio of commented tweets exceeds 20%. This makes sense, as the fake content posted by crowdturf workers may not be interesting enough for others to comment on. We also examine the number of people that each user has bidirectional comments with (bi-commentors). Crowdturf workers rarely interact with other users, with 66% of accounts having at most one bi-commentor.

Tweeting Clients. Next we look at the use of tweeting clients (devices). We can use the "device" field associated with each tweet to infer how tweets are sent. Tweet clients fall into four categories: web-based browsers, apps on mobile devices, third-party account management

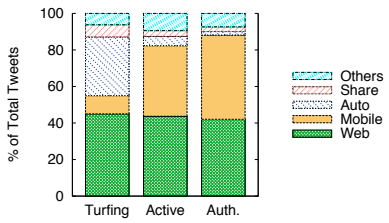


Figure 5: Tweet client usage.

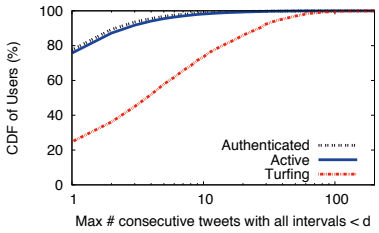


Figure 6: Max size of tweeting burst (threshold $d = 1$ minute).

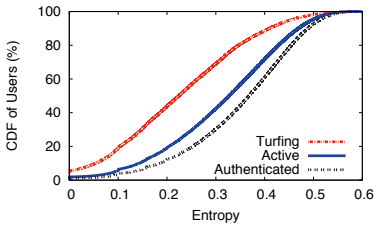


Figure 7: Normalized entropy of tweeting inter-arrival time.

Categ.	Top Tweet Clients
Web	Weibo Web, Weibo PC, 360Browser, Weibo Pro.
Mobile	iPhone, Android, iPad, XiaoMi
Auto	PiPi, Good Nanny, AiTuiBao, Treasure Box
Share	Taobao, Youku, Sina Blog, Baidu

Table 2: High-level categories for tweeting clients.

tools, and third-party websites via "share" buttons (Table 2). Figure 5 shows key differences in how different users use tweet clients. First, crowdturf workers use mobile (10%) much less than normal users (36% – 46%). One reason is that crowdturf workers rely on web browsers to interact with crowdturfing sites to get (submit) tasks and process payment, actions not supported by most mobile platforms.

We also observe that crowdturf workers are more likely to use automated tools. A close inspection shows that workers use these tools to automatically post non-spam tweets retrieved from a central content repository (*e.g.* a collection of hot topics). Essentially, crowdturf accounts use these generic tweets as cover traffic for their crowdturfing content. Third, crowdturf accounts "share" from third-party websites more often, since that is a common request in crowdturfing tasks [48].

Temporal Behavior. Finally, we look at temporal characteristics of tweeting behavior: tweet burstiness and periodicity. First, we expect task-driven workers to send many tweets in a short time period. We look for potential bursts, where each burst is defined as m consecutive tweets with inter-arrival times $< d$. We examine each user's maximum burst size (m) with different time thresholds d, *e.g.* Figure 6 depicts the result for d is set to 1 minute. We find that crowdturf accounts are more likely to post consecutive tweets within one-minute, something rarely seen from normal users. In addition, crowdturf workers are more likely to produce big bursts (*e.g.* 10 consecutive tweets with less than one-minute interval).

Second, workers accept tasks periodically, which can leave regular patterns in the timing of their tweets. We use *entropy* to characterize this regularity [16], where low entropy indicates a regular process while high entropy indicates randomness of tweeting. We treat each user's tweeting inter-arrival time as a random variable, and compute the first-order entropy [16]. Figure 7 plots user's entropy, normalized by the largest entropy in our dataset. Compared to normal users, crowdturf accounts in general have lower entropy, indicating their tweeting behaviors have stronger periodic patterns.

4 Detecting Crowdturfing Workers

We now use the features we identified to build a number of crowdturfing detectors using machine learning models. Here, we summarize the set of features we use for detection, and then build and evaluate a number of machine-learning detectors using our ground-truth data.

4.1 Key Features

We chose for our ML detectors a set of 35 features across five categories shown below.

- *Profile Fields (9).* We use 9 user profile fields[6] as features: follower count, followee count, followee-to-follower ratio, reciprocity, total tweet count, tweets per day, mentions per tweet, percent of tweets with mentions, and percent of tweets with embedded URLs.
- *User Interactions (8).* We use 8 features based on user interactions, *i.e.* comments and retweets. 4 features are based on user comments: percent of tweets with comments, percent of all comments that are outgoing, number of bi-commentors, and comment h-index (a user with h-index of h has at least h tweets each with at least h comments). We include 4 analogous retweet features.
- *Tweet Clients (5).* We compute and use the % of tweets sent from each tweet client type (web, mobile, automated tools, third-party shares and others) as a feature.

[6] Although profile fields *alone* cannot effectively detect crowdturf accounts (§3), they are still useful when combined with other features.

Alg.	Settings
NB	Default
BN	Default, K2 function
SVMr	Kernel γ =1, Cost parameter C =100
SVMp	Kernel degree d =3, Cost parameter C =50
J48	Confidence factor C =0.25, Instance/leaf M =2
RF	20 trees, 30 features/tree

Table 3: Classifier configurations.

- *Tweet Burstiness (12).* These 12 features capture the size and number of tweet bursts. A burst is m consecutive tweets where gaps between consecutive tweets are at most d minutes. For each user, we first compute the maximum burst size (m) while varying threshold d from 0.5 to 1, 30, 60, 120, 1440. Then we set d to 1 minute, and compute the number of bursts while varying size m over 2, 5, 10, 50, 100, and 500.
- *Tweeting Regularity (1).* This is the entropy value computed from each user's tweeting time-intervals.

4.2 Classification Algorithms

With these features, we now build classifiers to detect crowdturf accounts. We utilize a number of popular algorithms widely used in security contexts, including two Bayesian methods: Naive Bayesian (NB) [20] and BayesNet (BN) [18]; two Support Vector Machine methods [33]: SVM with radial basis function kernel (SVMr) and SVM with polynomial kernel (SVMp); and two Tree-based methods: C4.5 Decision Tree (J48[7]) [34] and Random Forests (RF) [7]. We leverage existing implementations of these algorithms in WEKA [17] toolkits.

Classifier and Experimental Setup. We start by constructing two experimental datasets, each containing all 28K turfing accounts, plus 28K randomly sampled baseline users from the "authenticated" and "active" sets. We refer to them as *Authenticated+Turfing* and *Active+Turfing*.

We use a small sample of ground-truth data to tune the parameters of different classifiers. At a high-level, we use grid search to locate the optimized parameters based on cross-validation accuracy. For brevity, we omit the details of the parameter tuning process and give the final configurations in Table 3. Note that features are normalized for SVM algorithms (we tested unnormalized approach which produced higher errors). We use this configuration for the rest of our experiments.

Basic Classification Performance. We run each classification algorithm on both experimental datasets with 10-fold cross-validation.[8] Figure 8 presents their classification error rates, including false positives (classifying normal users as crowdturf workers) and false negatives (classifying crowdturf accounts as normal users).

[7] J48 is WEKA's C4.5 implementation.

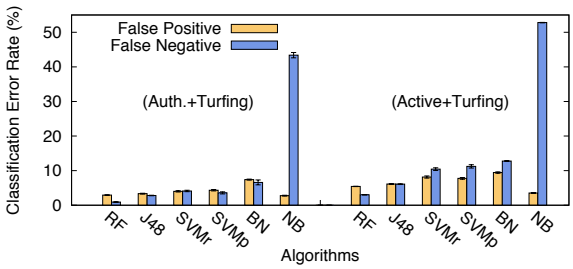


Figure 8: Classification error rates. Tree-based algorithms and SVMs outperform Bayesian methods.

We make four key observations. First, the two simple Bayesian methods generally perform worse than other algorithms. Second, Decision Tree (J48) and Random Forests (RF) are more accurate than SVMs. This is consistent with prior results that show SVMs excel in addressing high-dimension problems, while Tree algorithms usually perform better when feature dimensionality is low (35 in our case) [8]. Third, Random Forests outperform Decision Tree. Intuitively, Random Forests construct multiple decision trees from training data, which can more accurately model the behaviors of multiple types of crowdturf workers [7]. In contrast, decision tree would have trouble fitting distinct types of worker behaviors into a single tree. Finally, we observe that the two experiment datasets show consistent results in terms of relative accuracy across classifiers.

Comparing the two datasets, it is harder to differentiate crowdturf workers from *active* users than from *authenticated* users. This is unsurprising, since *authenticated* accounts often represent accounts of public figures, while *active* users are more likely to be representative of the normal user population. In the rest of the experiments, wherever the two datasets show consistent results, we only present the results on *Active+Turfing* for brevity, which captures the worse case accuracy for detectors.

4.3 Detecting Professional Workers

Our machine learning detectors are generally effective in identifying worker accounts. However, the contribution of tasks per worker is quite skewed, *i.e.* 90% of all tasks are completed by the top 10% most active "professional" workers (Figure 9). Intuitively, these "professional workers" are easier to detect than one-time workers. By focus-

[8] Cross-validation is used to compare the performance of different algorithms. We will split the data for training and testing the detectors later.

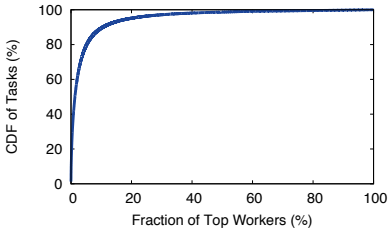


Figure 9: % of Tasks finished by top % of workers. The majority of spams were produced by top active workers.

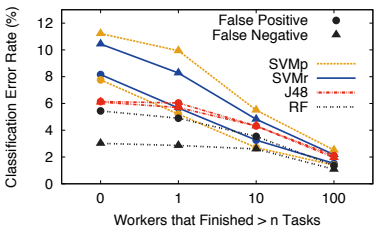


Figure 10: Classifying different levels of workers. Workers are filtered by # of crowdturfing tasks finished.

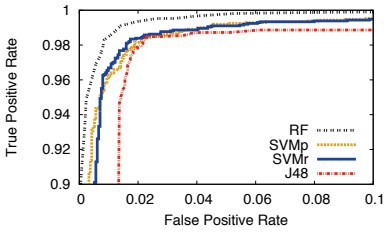


Figure 11: ROC curves of classifying professional workers (workers who finished more than 100 tasks).

ing on them, we can potentially improve detection accuracy while still effectively eliminate the largest majority of crowdturf output.

We evaluate classifier accuracy in detecting professional workers, by setting up a series of datasets each consisting of workers who performed more than n tasks (with n set to 1, 10, and 100). Each dataset also contains an equal number of randomly sampled normal users. We focus on the most accurate algorithms: Random Forests (RF), Decision Tree (J48) and SVM (SVMr and SVMp), and run 10-fold cross-validation on each of the datasets.

Figure 10 shows the classification results on *Active+Turfing*. As expected, our classifiers are more accurate in identifying "professional" workers. Different algorithms converge in accuracy as we raise the minimum productivity of professional workers. Accuracy is high for crowdturf workers who performed >100 tasks: Random Forests only produce 1.2% false positive rate and 1.1% false negative rate (99% accuracy). Note that while these top workers are only 8.9% of the worker population, they are responsible for completing 90% of all tasks. In the rest of the paper, we use "professional workers" to refer to workers who have completed >100 tasks.

False Positives vs. False Negatives. In practice, different application scenarios will seek different tradeoffs between false positives (FP) and false negatives (FN). For example, a system used as a pre-filter before more sophisticated tools (*e.g.* manual examination) will want to minimize FN, while an independent system without additional checks will want to minimize false positives to avoid hurting good users.

Figure 11 shows the ROC[9] curves of the four algorithms on the dataset of professional workers. Again, Random Forests perform best: they achieve extremely low false positive rate of <0.1% with only 8% false negative rate, or <0.1% false negative rate with only 7% false positive rate. We note that SVMs provide better false positive and false negative tradeoffs than J48, even though they have similar accuracy rates.

Imbalanced Data. We check our results on imbalanced data, since in practice there are more normal users than crowdturf workers. More specifically, we run our classifier (RF, professional) on imbalanced *testing* data with turfing-to-normal ratio ranging from 0.1 to 1. Note that we can still train our classifiers on balanced *training* data since we use supervised learning (we make sure training and testing data have no overlap). We find all the classifiers have accuracy above 98% (maximum FP 1.5%, FN 1.3%) against imbalanced testing data. We omit the plot for brevity.

Summary. Our results show that current ML systems can be used to effectively detect crowdturf workers. While this is a positive result, it assumes no adversarial response from the crowdturfing system. The following sections will examine detection efficacy under different levels of adversarial attacks.

5 Adversarial Attack: Evasion

We show that ML detectors can effectively identify "passive" crowdturf accounts in Weibo. In practice, however, crowdturfing adversaries can be highly adaptive: they will change their behaviors over time or can even intentionally attack the ML detectors to escape detection. We now re-evaluate the robustness of ML detectors under different adversarial environments, focusing on two types of adversaries:

1. *Evasion Attack:* individual crowd-workers adjust their behavior patterns to evade detection by trained ML detectors.
2. *Poisoning Attack:* administrators of crowdturfing sites participate, manipulating the ML detector training process by poisoning the training data.

We focus on evasion attacks in this section, and delay the study of poisoning attacks to §6. First, we define the evasion attack model. We then implement evasion attacks of different strengths, and study the performance

[9]ROC (receiver operating characteristic) is a plot that illustrates classifier's false positives and true positives versus detection threshold.

of ML detectors accordingly. Specifically, we consider "optimal evasion" attacks, where adversaries have full knowledge about the ML detectors and the Weibo system, and more "practical" evasion attacks, where adversaries have limited knowledge about the detectors and the Weibo system.

5.1 Basic Evasion Attack Model

Evasion attacks refer to individual crowdturfing workers seeking to escape detection by altering their own behavior to mimic normal users. For example, given knowledge of a deployed machine learning classifier, a worker may attempt to evade detection by selecting a subset of user features, and replacing their values with the *median* of the observed normal user values. Since mimicking normal users reduces crowdturfing efficiency, workers are motivated to minimize the number of features they modify. This means they need to identify a minimal core set of features enabling their detection.[10]

This attack makes two assumptions. *First*, it assumes that adversaries, *i.e.* workers, know the list of features used by the classifiers. Technical publications, *e.g.* on spam detection [4, 43, 50], make it possible for adversaries to make reasonable guesses on the feature space. *Second*, it assumes that adversaries understand the characteristics of normal users in terms of these features. In practice, this knowledge can be obtained by crawling a significant portion of Weibo accounts.

Depending on their knowledge of the ML features and of normal user behavior, adversaries can launch evasion attacks of different strengths. We implement and evaluate ML models on a range of threat models with varying levels of adversary knowledge and computational capabilities. We start from the *optimal evasion* scenario, where adversaries have *complete* knowledge of the feature set. The corresponding ML detector results represent worst-case performance (or lower bound) under evasion attacks. We also study a set of *practical evasion* models where adversaries have limited (and often noisy) knowledge, and constrained resources.

5.2 Optimal Evasion Attack

In this *ideal* case, adversaries have perfect knowledge about the set of features they need to modify. To understand the impact of the feature choices, we look at multiple variants of the optimal evasion models. These include the *per-worker optimal evasion model*, where each worker finds her own optimal set of features to alter, the *global optimal evasion* where all workers follow the same optimal set of features to alter, and *feature-aware evasion* where workers alter the most important features.

[10] For simplicity, we consider features to be independent.

We implement these evasion models on our ground-truth dataset, and evaluate ML detector accuracy. Note that these attacks we identify are not necessarily practical, but are designed to explore worse-case scenarios for ML models.

Per-worker Optimal Evasion. Intuitively, each worker should have her own optimal strategy to alter features, *e.g.* some workers need to add followers first, while others need to reduce tweeting burstiness. Doing so is hard in practice: each worker has to apply exhaustive search to identify its optimal strategy that minimizes the set of features to modify.

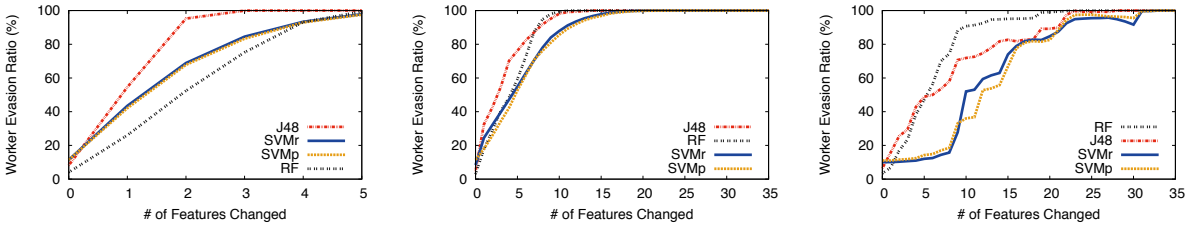
We implement this scenario on our *Active+Turfing* dataset. We first split the data into equal-sized training and testing datasets, and use the top-4 most accurate algorithms to build classifiers with authentic training data. We then run detection on worker accounts in the testing dataset. Here for each worker, we exhaustively test all combinatorial combinations of possible features to modify until the classifier classifies this worker as normal. In this way, we find the minimal set of features each user must modify to avoid detection.

Figure 12(a) plots the evasion rate for the four ML algorithms. Clearly, this optimal evasion model is highly effective. By simply altering one feature, 20-50% of workers can evade detection (different workers can choose to alter different features). And by altering five features, 99% of workers can evade all four classifiers. We also observe that the Random Forests (RF) algorithm achieves the best robustness, since it requires the most number of features to be altered.

Global Optimal Evasion. The per-worker model makes a strong assumption that each worker can identify her own optimal feature set. Next, we loosen this assumption and only assume that all workers exercise a uniform strategy. This is possible if a third-party (*e.g.* site admin) guides workers in altering their features.

To identify the global optimal strategy, we search exhaustively through all possible feature combinations, and locate the feature set (for a given size) that allows the majority of workers to achieve evasion. The corresponding evasion rate result is in Figure 12(b). 99% of workers can successfully evade all four detectors by altering 15 features, which is much larger than the per-worker case (5 features). This is because any one-size-fits-all strategy is unlikely to be ideal for individual workers, thus the feature set must be large enough to cover all workers.

Feature-aware Evasion. Achieving optimal evasion is difficult, since it requires adversaries to have knowledge of the trained classifiers. Instead, this model assumes that adversaries can accurately identify the relatively "importance" of the features. Thus workers alter the most important features to try to avoid detection.



(a) Per-worker Optimal Evasion (b) Global Optimal Evasion (c) Feature Importance Aware Evasion

Figure 12: Evasion rate of optimal evasion strategies (all workers).

We implement this attack by building the classifiers and then computing the feature importance. For this we use the χ^2 (Chi Squared) statistic [51], a classic metric to measure feature's discriminative power in separating data instances of different classes[11]. During detection, workers alter features based on their rank.

Figure 12(c) plots evasion results for the four classifiers. We make two key observations. First, this feature-aware strategy is still far away from the per-worker optimal case (Figure 12(a)), mostly because it is trying to approximate global optimal evasion. Second, performance depends heavily on the underlying classifier. For RF and J48, performance is already very close to that of the global optimal case, while the two SVM algorithms are more resilient. A possible explanation is that the χ^2 statistic failed to catch the true feature importance for SVM, since SVM normalizes feature values before training the classifier. These results suggest that without knowing the specific ML algorithm used by the defenders, it is hard to avoid detection even knowing the importance of features.

5.3 Evasion under Practical Constraints

Our results show workers can evade detection given complete knowledge of the feature set and ML classifiers. However, obtaining complete knowledge is very difficult in practice. Thus we examine *practical* evasion threat models to understand their efficacy compared to optimal evasion models. We identify practical constraints facing adversaries, present several practical threat models and evaluate their impact on our detectors.

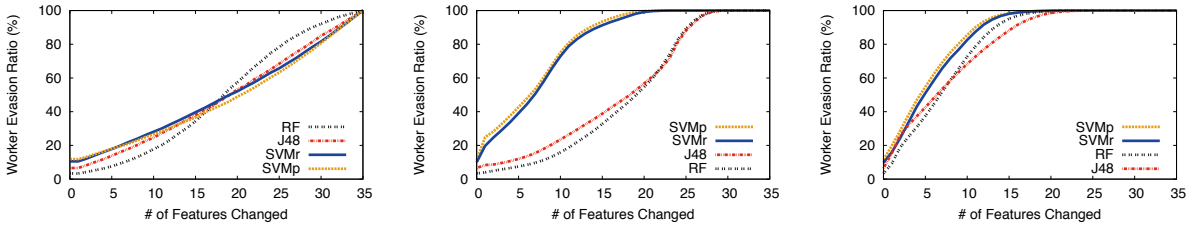
Practical Constraints. In practice, adversaries face two key resource constraints. First, they cannot reverse-engineer the trained classifier (*i.e.* the ML algorithm used or its model parameters) by querying the classifier and analyzing the output – it is too costly to establish millions of profiles with controlled features and wait for some of them to get banned. Thus workers cannot perform exhaustive search to launch optimal evasion attacks, but have to reply on their partial knowledge for evasion. Second, it is difficult for adversaries to obtain *complete* statistics of normal users. They can estimate normal user statistics via a (small) sampling of user profiles, but estimation errors are likely to reduce their ability to precisely mimic normal users.

Next, we will examine each constraint separately, and evaluate the likely effectiveness of attacks under the more realistic conditions.

Distance-aware Evasion. We consider the first constraint which forces workers to rely on *partial* knowledge to guide their evasion efforts. In this case, individual workers are only aware of their own accounts and normal user statistics. When choosing features to alter, they can prioritize features with the largest differential between them and normal users. They must quantify the "distance" between each crowdturf account and normal users on a given feature. Here, we pick two very intuitive distance metrics and examine the effectiveness of the corresponding evasion attacks. For now, we ignore the second constraint by assuming workers have perfect knowledge of average user behaviors.

- *Value Distance (VD):* Given a feature k, this captures the distance between a crowd-worker i and normal user statistics by $VD(i,k) = \frac{|F_k(i) - Median(N_k)|}{Max(N_k) - Min(N_k)}$ where $F_k(i)$ is the value of feature k in worker i, and N_k is normal user statistical distribution on feature k. When altering feature k, worker i replaces $F_k(i)$ with $Median(N_k)$.
- *Distribution Distance (DD):* Here the distance depends on where $F_k(i)$ is positioned within N_k. For example, if $F_k(i)$ is around 50%-tile of N_k, then worker i is similar to a normal user. Therefore, we define the distance by $DD(i,k) = |Percentile(N_k, F_k(i)) - 50|/100$ where $Percentile(N_k, F_k(i))$ is the percentile of $F_k(i)$ in the normal user CDF N_k. Note that when $F_k(i)$ exceeds the range of N_k, this distance metric becomes invalid. However, our data suggests that this rarely happens ($<1\%$).

[11] We also tested information gain to rank features, which produced similar ranking results (*i.e.* the same top-10 as using χ^2).



(a) Random Evasion Strategy (Random) (b) Value Distance Aware Strategy (VD) (c) Distribution Distance Aware Strategy (DD)

Figure 13: Evasion rate of practical evasion strategies (all workers).

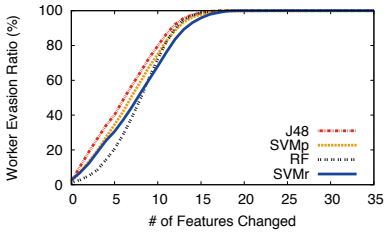


Figure 14: Evasion rate using distribution distance aware strategy (DD) for professional workers.

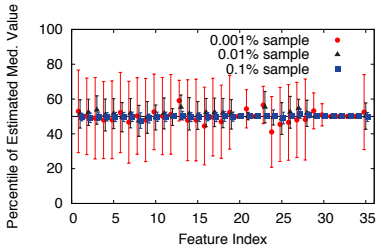


Figure 15: The percentile of *estimated* median value in the true normal user CDF.

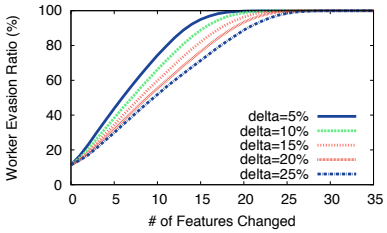


Figure 16: Impact of median value estimation error on evasion rate, using DD evasion on SVMp.

To evaluate the impact of practical evasion attacks, we split the *Active+Turfing* data into equal-sized training and testing sets. After classifier training, we simulate the distance-aware evasion attacks on the testing data. Figure 13(b) and 13(c) show evasion rates based on VD and DD respectively. As a baseline, we also show Figure 13(a) where adversaries *randomly* select features to alter. Compared to random evasion, distance-based approaches require much less feature altering. For example, when altering 15 features, random approach only saves <40% of workers, while distance strategies provide as high as 91% (VD-SVMp) and 98% (DD-SVMp).

The four classifiers perform very differently. RF and J48 classifiers are much more vulnerable to DD based evasion than to VD based evasion. While SVMs perform similarly in both strategies. In general, Tree-based algorithms are more robust than SVM classifiers against distance-aware evasions. This is very different to what we observed in the optimal evasion cases (Figure 12(a)–12(b)), where SVMs are generally more robust. This suggests that theoretical bounds on ML algorithms may not truly reflect their performance in practice.

Consistently, the impact of practical evasion attacks is much weaker than that of optimal evasion (*i.e.* per-worker optimal). Adversaries are severely constrained by lack of knowledge of detection boundaries of the classifiers, and have to guess based on "distance" information. The implication is that the less adversaries know about classifiers, the harder it is for them to evade detection.

We also evaluate the attack impact on classifiers to detect professional workers. We find the general trends are similar and only show the results of DD-based attack in Figure 14. We note that it is easier to evade classifiers dedicated to detect professionals (compared with Figure 13(c)). This is because when trained to a smaller, more homogeneous worker population, classifiers expect strong malicious behaviors from crowd-workers. Thus even a small deviation away from the model towards normal users will help achieve evasion.

Impact of Normal User Estimation Errors. We extend the above model by accounting for possible errors in estimating normal user behaviors (the second constraint). These errors exist because adversaries can only sample a limited number of users, leading to noisy estimations. Here, we investigate the impact of sampling strategies on the attack efficacy.

For all 35 features, we vary the sampling rate, *i.e.* the ratio of normal users sampled by adversaries, by taking random samples of 0.001%, 0.01% to 0.1% of the *Active* dataset. We repeat each instance 100 times, and compute the mean and standard deviation of the estimated median feature values (Figure 15). We show each feature's *percentile* in the true CDF of the *Active* dataset. In this case, the optimal value is 50%. Clearly sampling rate does impact feature estimation. With the 0.001% sampling rate, the estimated value varies significantly across instances. Raising the sample rate to 0.1% means attackers can accurately estimate the median value using only

a few instances. Furthermore, we see that burstiness features (*e.g.* features 30-34) are easy to sample, since normal user values are highly skewed to zero.

Finally, we evaluate the impact of estimation errors on practical evasion attacks. This time we run distance-aware evasions based on the *estimated* median feature values. For each worker's feature k, we estimate the median value $M'(k)$ with a given bound of error Δ. That is, $M'(k)$ is randomly picked from the percentiles within $[50\% - \Delta, 50\% + \Delta]$ on the true CDF of normal user behaviors. By iterating through different Δ (from 5% to 25%), our results show that Δ only has a minor impact. The most noticeable impact is on SVMp using DD distance (Figure 16). Overall, we conclude that as long as adversaries can get a decent guess on normal user behaviors, the residual noise in the estimation Δ should not affect the efficacy of evasion attacks.

Summary. Our work produces two key observations.

- Given complete knowledge, evasion attacks are very effective. However, adversaries under more realistic constraints are significantly less effective.
- While no classifier is robust against all attack scenarios, there is a consistent inverse relationship between single model fitting accuracy and robustness to adversarial evasion. Highly accurate fit to a smaller, more homogeneous population (*e.g.* professionals) makes models more vulnerable to evasion attacks.

6 Adversarial Attack: Poisoning

After examining evasion attacks, we now look at how centralized crowdturfing sites can launch more powerful attacks to manipulate machine learning models. Specifically, we consider the poisoning attack where administrators of crowdturfing sites intentionally pollute the training dataset used to build ML classifiers, forcing defenders to produce inaccurate classifiers. Since the training data (*i.e.* crowdturfing accounts) actually comes from these crowdturfing sites, administrators are indeed capable of launching these attacks.

In the following, we examine the impact of poisoning attacks on ML detection accuracy. We consider two mechanisms for polluting training data. The first method directly adds misleading/synthetic samples to the training set. Adversaries in the second method simply alter data records, or modify operational policies to alter the composition of the training data used by ML models.

6.1 Injecting Misleading Samples

Perhaps the simplest way to pollute any training data is to add misleading or false samples. In our case, since the training data has two classes (groups) of accounts, this can be done by mixing normal user samples into the "turfing" class, *i.e.* poisoning the turfing class, or mixing crowdturf samples into the "normal" user class, *i.e.* poisoning the normal class. Both introduce incorrectly labeled training data to mislead the classifier.

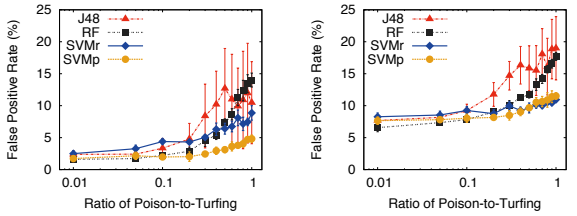
Poisoning Turfing Class. To poison the turfing class, adversaries (*e.g.* ZBJ and SDH administrators) add normal Weibo accounts to the public submission records in their own systems. Since ML classifiers take ground-truth crowdturf accounts from those public records, these benign accounts will then be mixed into the training data and labeled as "turfing." The result is a model that marks some characteristics of normal users as crowdturfing behavior, likely increasing false positive rate in detection.

We simulate the attack with our ground-truth dataset. At a high level, we train the classifiers on "polluted" training data, and then examine changes in classifiers' detection accuracy. Here we experiment with two strategies to pollute the turfing class. First, as a baseline strategy, adversaries *randomly* select normal users as poison samples to inject into the turfing class. Second, adversaries can inject *specific* types of normal users, causing the classifiers to produce *targeted* mistakes.

Random Poisoning: We simulate this poisoning attack with *Active+Turfing* dataset, where adversaries inject random normal accounts to the turfing class. Specifically, for training, the turfing class (14K accounts) is a mixture of crowdturf accounts and poison samples randomly selected from *Active*, with a mixing ratio of p. The normal class is another 14K normal accounts from *Active*. Then we use 28K of the rest accounts (14K turfing and 14K normal users) for testing. For any given p, we repeat the experiment 10 times with different random poison samples and training-testing partitions to compute average detection rates.

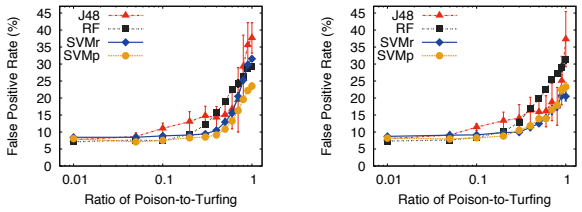
Results are shown in Figure 17(b). As a baseline comparison, we also present the results of the classifiers for professional workers in Figure 17(a). We have three observations. First, as poison-to-turfing ratio p increases, false positive rates go up for all four algorithms. False negative rates are not much affected by this attack, thus are omitted from the plot.[12] Second, we find that the SVM classifiers are more resilient: SVMp's false positive rate increases <5% as p approaching 1.0, while the analogous increases exceed 10% for Random Forests and J48. Particularly, J48 experiences more drastic fluctuations around average, indicating it is very sensitive to the choice of poison samples. This is consistent with our prior observation that more accurate single model fitting (*i.e.* J48 is more accurate than SVM) is more vulnerable to adversarial attacks. Similarly, highly accurate detec-

[12] False negative rates increase $< 2\%$ when p approaches 1.0.



(a) Professional Workers (b) All Workers

Figure 17: Poisoning training dataset by injecting random normal user samples to the turfing class.



(a) Injecting Accounts with $>50\%$ tweets commented (b) Injecting Accounts with <150 followers

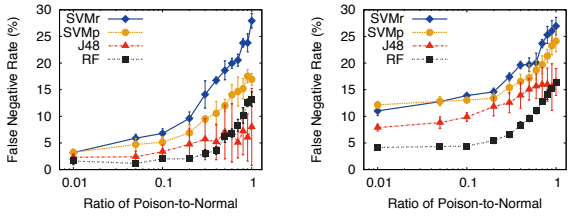
Figure 18: Targeted poisoning. Adversaries inject specific type of normal users to the turfing class (all workers).

tion of the more homogeneous population of professional workers (§4) means the models experience larger relative impacts from attacks compared to classifiers over all workers.

Note that we limited the poison-to-turfing ratio <1, since in practice adversaries cannot inject unlimited poison samples to defender's training data. First, injecting noise causes inconvenience to their own customers in locating qualified workers. Second, defenders may collect ground-truth records from multiple crowdturfing sites.

Targeted Poisoning: Next, we explore *targeted* poisoning to the turfing class. Here the adversaries want to carefully inject selected poison samples so classifiers make targeted mistakes. For example, our classifier uses "ratio of commented tweets" as a feature with the intuition that worker's tweets rarely receive comments (§3). Once adversaries gain this knowledge, they can intentionally select accounts whose tweets often receive comments as the poison samples. As a result, the trained classifier will mistakenly learn that users with high comment ratio can be malicious, thus are likely to misclassify this kind of normal users as crowd-workers.

To evaluate the impact of targeted poisoning, we perform similar experiments, except that we select poison samples based on specific feature. Figure 18 shows the attacking results on two example features: ratio of tweets with comments and follower count. Compared with Figure 17, targeted poisoning can trigger higher false positives than randomly selecting poison samples. Also, the previous observations still hold with SVM being more robust and J48 experiencing unstable performance (large deviation from average).



(a) Professional Workers (b) All Workers

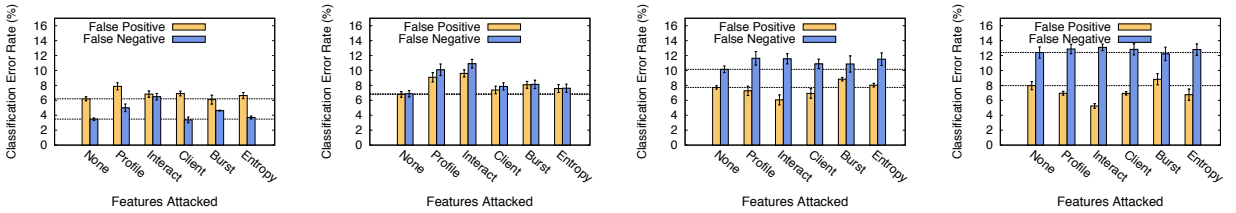
Figure 19: Poisoning training dataset by adding turfing samples to normal class.

Poisoning Normal User Class. Next, we analyze the other direction where adversaries inject turfing samples into the "normal" class to boost the *false negative rate* of classifiers. This may be challenging in practice since the normal user pool – Weibo's whole user population – is extremely large. Hence it requires injecting a significant amount of misleading samples in order to make an impact. Here from defender's perspective, we aim to understand how well different classifiers cope with "noisy" normal user data.

We repeat the previous "Random Poisoning" attack on the normal class. Figure 19(a) and Figure 19(b) show the attack results on classifiers for professional workers and all workers respectively. As we increase the ratio of poison samples, the false negatives of all four classifiers increase. This is expected as the classifiers will mistakenly learn crowdturf characteristics when modeling normal users, thus are likely to misclassify turfing accounts as benign later. In addition, we find the robustness of different classifiers varies, with Random Forests algorithm producing the lowest overall false negatives. Finally, we again observe that the more accurate classifier for professional workers suffers larger relative impacts from adversaries than classifiers for all-workers.

6.2 Altering Training Data

The above poisoning attacks focus on misleading classifiers to catch the wrong target. However, it does not fundamentally prevent crowd-workers from detection, since workers' behavior patterns are still very differently from normal users. To this end, we explore a second poisoning attack, where adversaries directly *alter* the training data by tuning crowd-workers' behavior to mimic normal users. The goal is to make it difficult (or even impossible) to train an accurate classifier that isolates crowdturf accounts with normal accounts.



(a) Random Forests (b) J48 (c) SVMr (d) SVMp

Figure 20: Performance of different classifiers when adversaries alter crowd-workers' features to mimic normal users. The horizontal lines represent the baseline false positive (false negative) rates when no feature is altered.

To carry out this attack, adversaries (*e.g.* administrators of ZBJ and SDH) need to modify the behaviors of numerous crowdturf workers. This can be done by centrally enforcing operational policies to their own system. For example, enforcing minimal time interval between taking tasks to reduce the tweeting burstiness or enforcing screening mechanisms to reject worker accounts with "malicious" profile features. In the following, we evaluate the attack impact using simulations, followed by the discussion of practical costs.

Feature Altering Attack. To simulate this attack, we let adversaries select a set of features F of crowdturf accounts and alter F to mimic the corresponding features of normal users. Unlike evasion attacks that can simply mimic normal users' median values, here we need to mimic the whole distribution in order to make the two classes indistinguishable on these features. Since the feature altering is for all workers in the crowdturfing system, thus it actually applies to crowdturf accounts in both training and testing datasets. Finally, note that features are not completely independent, *i.e.* changing one feature may cause changes in others. To mitigate this, we tune features under the same category simultaneously.

Figure 20 shows the attack results on *Turfing+Active* dataset. We attack each feature category and repeat the experiment for 10 times. Here we simulate attacking one category at a time, and will discuss attacking category combinations later. In general, the attack makes all classifiers produce higher error rates compared with baseline where no feature is altered (the horizontal lines). However the impact is mild compared to injection-based poisoning attacks. For example, the most effective attack is on J48 when altering interaction features, which causes error rate increased by 4%, while injection-based attack can boost error rate by more than 20% (Figure 18). One possible reason is that unlike injection-based poisoning, altering-based poisoning does not cause inconsistencies in training and testing data, but only make the two classes harder to separate.

Costs of Altering. In practice, feature altering comes with costs, and some features may be impossible to manipulate even by crowdturfing administrators. For instance, *Tweeting Regularity* (Entropy) and *Burstiness* features are easier to alter. Recall that crowdturfing systems can enforce minimal (random) time delay between workers taking on new tasks, or use delays to increase entropy. Changing the *Tweet Client* feature is also possible, since crowdturfing systems can develop mobile client software for their workers, or simply release tools for workers to fake their tweeting clients.

Features Attacked	Error Rate (FP %, FN %)			
	RF	**J48**	**SVMr**	**SVMp**
None	(6.2, 3.4)	(6.7, 6.8)	(7.7, 10.1)	(7.9, 12.1)
C+B	(5.7, 4.4)	(7.9, 8.7)	(8.7, 12.2)	(8.0, 14.0)
B+E	(6.5, 3.9)	(7.1, 7.8)	(8.7, 12.5)	(7.3, 13.1)
C+E	(6.4, 4.5)	(7.9, 8.2)	(7.5, 11.8)	(6.3, 13.8)
C+B+E	(5.8, 4.2)	(8.3, 8.5)	(8.6, 13.2)	(7.7, 15.2)

Table 4: Error rates when features are altered in combinations. We focus on attacking low-cost features: Tweet Client (C), Burstiness (B) and Entropy (E).

Profile and *Interaction* features are more difficult to alter. Some features are mandatory for common tasks. For example, workers need to maintain a certain number of followers in order to spread spam to reach large enough audiences. In addition, some features are rooted in the fact that crowd-workers don't use their accounts organically, which, making it hard to generate normal user interactions. Although, crowdturfing systems could potentially use screening mechanisms to reject obviously-malicious crowdturf accounts from their system. However, this high bar will greatly shrink the potential worker population, and likely harm the system's spam capacity.

Considering practical costs, we consider whether it is more impactful to alter the combinations of features from different categories. Here we focus on altering the low cost features in *Tweet Client* (C), *Burstiness* (B) and *Entropy* (E). As shown in Table 4, attacking feature combinations produces slightly higher error rates than attacking a single feature category, but the overall effect is still small (less than 4% error rate increase).

Summary and Discussion. Through our analysis, we find that injecting misleading samples into training data causes more significant errors than uniformly altering worker behavior. In addition, we again observe the inverse relationship between single model fitting accuracy and robustness.

To protect their workers, crowdturfing sites may also try to apply stronger access control to their public records in order to make training data unavailable for ML detectors[13]. However, this creates obvious inconvenience for crowdturfing sites, since they rely on these records to attract new workers. Moreover, even if records were private, defenders can still obtain training data by joining as "customers," offering tasks, and identifying accounts of participating workers.

7 Related Work

Crowdturfing. Prior works used measurements on crowdturfing sites to understand their operation and economic structure [23, 24, 26, 48]. Some systems have been developed to detect paid human spammers in online review sites [31] and Q&A systems [9, 45]. To the best of our knowledge, our work is the first to explore detection of crowdturfing behaviors in adversarial settings.

OSN Spammer Detection. Researchers have developed mechanisms to detect fake accounts (Sybil) and spam campaigns in online social networks, including Facebook [15, 49], Twitter [43], Renren [52] and LinkedIn [46]. Most prior works develop ML models using features of spammer profiles (*e.g.* FFRatio, blacklisted URLs) or bot-like behaviors [4, 11, 42, 47, 50]. However, a recent study shows dedicated spam bots can still infiltrate social networks without being detected [14]. In our case, crowdturf accounts are carefully maintained by human users, and their questionable activities are camouflaged with synthetic cover traffic. This makes their detection challenging, until we add additional behavioral features (*e.g.* user-interaction, task-driven behavior).

Adversarial Machine Learning. In an early study [19], researchers classify ML adversarial attacks into two high-level categories: *causative* attacks where adversaries alter the training process to damage the classifier performance, and *exploratory* attacks where adversaries try to circumvent an already-trained classifier. Much of existing work focuses on *exploratory* attacks [5, 12, 25, 28] with less focusing on *causative* attacks [6, 37], since it's usually more difficult for adversaries to access training data in practice. In this paper, we studied both angles as both attacks are practically feasible from crowdturfing adversaries.

Several studies have examined attacks on specific ML-based applications, from email spam detection [12] to network intuition detection [37, 40, 44] to malicious (PDF) file classification [5, 25, 41] and malware detection [21]. Our work focuses on crowdturfing and explores a wider range of adversarial attacks, including active evasion and more powerful poison attacks against the model training process.

8 Conclusion and Discussion

We use a large-scale ground truth dataset to develop machine learning models to detect malicious crowdsourcing workers. We show that while crowdturfing workers resemble normal users in their profiles, ML models can effectively detect regular workers (95% accuracy) or "professionals" (99% accuracy) using distinguishing features such as user interactions and tweet dynamics.

More importantly, we use crowdturfing defense as context to explore the robustness of ML algorithms against adversarial attacks. We evaluate multiple adversarial attack models targeting both training and testing phases of ML detectors. We find that these attacks are effective against all machine learning algorithms, and coordinated attacks (such as those possible in crowdturfing sites) are particularly effective. We also note a consistent tradeoff where more accurate fits (especially to a smaller, more homogeneous population) result in higher vulnerability to adversarial attacks. The exception appears to be Random Forests, which often achieves both high accuracy and robustness to adversaries, possibly due to its natural support for multiple populations.

Limitations and Future Work. We note that our study has several limitations. First, our analysis focuses on Weibo, and our adversary scenarios may not generalize fully to other platforms (*e.g.* review sites, instant message networks). However, more work is necessary to validate our findings on other platforms. Second, our adversarial models use simplifying assumptions, *i.e.* features are independent and costs for feature modification are uniform. In addition, attackers may behave differently to disrupt the operation of ML detectors.

Moving forward, one goal is to validate our adversarial models in practice, perhaps by carrying out a user-study on crowdturfing sites where we ask workers to actively evade and disrupt ML detectors. In addition, our results show we must explore approaches to improve the robustness of ML-based systems. Our analysis showed that ML algorithms react differently to different adversarial attacks. Thus one possible direction is to develop hybrid systems that integrate input from multiple classi-

[13] As of late 2013, some crowdturfing sites (*e.g.* ZBJ) have already started to follow this direction, by limiting access to public transaction records to verified active participants.

fiers, ideally without affecting overall accuracy. We also observe that limiting adversaries' knowledge of the target system can greatly reduce their attack abilities. How to effectively prevent adversaries from gaining knowledge or reverse-engineering models is also a topic for future work.

9 Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback, and Xifeng Yan for insightful discussions. This work is supported in part by NSF grants IIS-1321083, CNS-1224100, IIS-0916307, by the DARPA GRAPHS program (BAA-12-01), and by the Department of State. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Sina Weibo Admin Statement on Spam. `http://www.weibo.com/p/1001603697836242954625`, April 2014.

[2] Sina Weibo Terms of Service. `http://service.account.weibo.com/roles/guiding`, 2014. (The link is accessible after login).

[3] AHN, Y.-Y., HAN, S., KWAK, H., MOON, S., AND JEONG, H. Analysis of topological characteristics of huge online social networking services. In *Proc. of WWW* (2007).

[4] BENEVENUTO, F., MAGNO, G., RODRIGUES, T., AND ALMEIDA, V. Detecting spammers on twitter. In *Proc. of CEAS* (2010).

[5] BIGGIO, B., CORONA, I., MAIORCA, D., NELSON, B., SRNDIC, N., LASKOV, P., GIACINTO, G., AND ROLI, F. Evasion attacks against machine learning at test time. In *Proc. of ECML PKDD* (2013).

[6] BIGGIO, B., NELSON, B., AND LASKOV, P. Poisoning attacks against support vector machines. In *Proc. of ICML* (2012).

[7] BREIMAN, L. Random forests. *Machine Learning 45*, 1 (2001), 5–32.

[8] CARUANA, R., KARAMPATZIAKIS, N., AND YESSENALINA, A. An empirical evaluation of supervised learning in high dimensions. In *Proc. of ICML* (2008).

[9] CHEN, C., WU, K., SRINIVASAN, V., AND R, K. B. The best answers? think twice: Online detection of commercial campaigns in the cqa forums. *CoRR* (2012).

[10] CHEN, X. Dairy giant mengniu in smear scandal. China Daily, October 2010.

[11] CHU, Z., GIANVECCHIO, S., WANG, H., AND JAJODIA, S. Who is tweeting on twitter: human, bot, or cyborg? In *Proc. of ACSAC* (2010).

[12] DALVI, N., DOMINGOS, P., MAUSAM, SANGHAI, S., AND VERMA, D. Adversarial classification. In *Proc. of KDD* (2004).

[13] EATON, K. Mechanical turk's unsavory side effect: Massive spam generation. Fast Company, December 2010.

[14] FREITAS, C. A., BENEVENUTO, F., GHOSH, S., AND VELOSO, A. Reverse engineering socialbot infiltration strategies in twitter. *CoRR abs/1405.4927* (2014).

[15] GAO, H., HU, J., WILSON, C., LI, Z., CHEN, Y., AND ZHAO, B. Y. Detecting and characterizing social spam campaigns. In *Proc. of IMC* (2010).

[16] GIANVECCHIO, S., AND WANG, H. Detecting covert timing channels: an entropy-based approach. In *Proc. of CCS* (2007).

[17] HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. The weka data mining software: an update. *SIGKDD Explor. Newsl. 11*, 1 (2009).

[18] HECKERMAN, D., GEIGER, D., AND CHICKERING, D. M. Learning bayesian networks: The combination of knowledge and statistical data. *Mach. Learn. 20*, 3 (1995), 197–243.

[19] HUANG, L., JOSEPH, A. D., NELSON, B., RUBINSTEIN, B. I., AND TYGAR, J. D. Adversarial machine learning. In *Proc. of AISec* (2011).

[20] JOHN, G. H., AND LANGLEY, P. Estimating continuous distributions in bayesian classifiers. In *Proc. of UAI* (1995).

[21] KANTCHELIAN, A., AFROZ, S., HUANG, L., ISLAM, A. C., MILLER, B., TSCHANTZ, M. C., GREENSTADT, R., JOSEPH, A. D., AND TYGAR, J. D. Approaches to adversarial drift. In *Proc. of AISec* (2013).

[22] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing network-wide traffic anomalies. In *Proc. of SIGCOMM* (2004).

[23] LEE, K., TAMILARASAN, P., AND CAVERLEE, J. Crowdturfers, campaigns, and social media: Tracking and revealing crowdsourced manipulation of social media. In *Proc. of ICWSM* (2013).

[24] LEE, K., WEBB, S., AND GE, H. The dark side of micro-task marketplaces: Characterizing fiverr and automatically detecting crowdturfing. In *Proc. of ICWSM* (2014).

[25] MAIORCA, D., CORONA, I., AND GIACINTO, G. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proc. of ASIACCS* (2013).

[26] MOTOYAMA, M., MCCOY, D., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. Dirty jobs: The role of freelance labor in web service abuse. In *Proc. of Usenix Security* (2011).

[27] NELSON, B., RUBINSTEIN, B. I. P., HUANG, L., JOSEPH, A. D., HON LAU, S., LEE, S. J., RAO, S., TRAN, A., AND TYGAR, J. D. Near-optimal evasion of convex-inducing classifiers. In *Proc. of AISTATS* (2010).

[28] NELSON, B., RUBINSTEIN, B. I. P., HUANG, L., JOSEPH, A. D., LEE, S. J., RAO, S., AND TYGAR, J. D. Query strategies for evading convex-inducing classifiers. *J. Mach. Learn. Res. 13*, 1 (2012).

[29] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. of IEEE S&P* (2005).

[30] ONG, J. China's sina weibo grew 73% in 2012, passing 500 million registered accounts. The Next Web, Feb. 2013.

[31] OTT, M., CHOI, Y., CARDIE, C., AND HANCOCK, J. T. Finding deceptive opinion spam by any stretch of the imagination. In *Proc. of ACL* (2011).

[32] PHAM, N. Vietnam admits deploying bloggers to support government. BBC News, January 2013.

[33] PLATT, J. C. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning* (1998).

[34] QUINLAN, J. R. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[35] RAMACHANDRAN, A., FEAMSTER, N., AND VEMPALA, S. Filtering spam with behavioral blacklisting. In *Proc. of CCS* (2007).

[36] ROBINSON, G. A statistical approach to the spam problem. *Linux J. 2003*, 107 (2003).

[37] RUBINSTEIN, B. I., NELSON, B., HUANG, L., JOSEPH, A. D., LAU, S.-H., RAO, S., TAFT, N., AND TYGAR, J. D. Antidote: understanding and defending against poisoning of anomaly detectors. In *Proc. of IMC* (2009).

[38] SHEAR, M. D. Republicans use crowdsourcing to attack obama campaign. The New York Times, May 2012.

[39] SIMONITE, T. Hidden industry dupes social media users. MIT Review, December 2011.

[40] SOMMER, R., AND PAXSON, V. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of IEEE S&P* (2010).

[41] SRNDIC, N., AND LASKOV, P. Practical evasion of a learning-based classifier: A case study. In *Proc. of IEEE S&P* (2014).

[42] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Detecting spammers on social networks. In *Proc. of ACSAC* (2010).

[43] THOMASY, K., GRIERY, C., PAXSONY, V., AND SONGY, D. Suspended accounts in retrospect: An analysis of twitter spam. In *Proc. of IMC* (2011).

[44] VENKATARAMAN, S., BLUM, A., AND SONG, D. Limits of learning-based signature generation with adversaries. In *Proc. of NDSS* (2008).

[45] WANG, G., GILL, K., MOHANLAL, M., ZHENG, H., AND ZHAO, B. Y. Wisdom in the social crowd: an analysis of quora. In *Proc. of WWW* (2012).

[46] WANG, G., KONOLIGE, T., WILSON, C., WANG, X., ZHENG, H., AND ZHAO, B. Y. You are how you click: Clickstream analysis for sybil detection. In *Proc. of USENIX Security* (2013).

[47] WANG, G., MOHANLAL, M., WILSON, C., WANG, X., METZGER, M., ZHENG, H., AND ZHAO, B. Y. Social turing tests: Crowdsourcing sybil detection. In *Proc. of NDSS* (2013).

[48] WANG, G., WILSON, C., ZHAO, X., ZHU, Y., MOHANLAL, M., ZHENG, H., AND ZHAO, B. Y. Serf and turf: crowdturfing for fun and profit. In *Proc. of WWW* (2012).

[49] WILSON, C., SALA, A., PUTTASWAMY, K. P. N., AND ZHAO, B. Y. Beyond social graphs: User interactions in online social networks and their implications. *ACM Transactions on the Web 6*, 4 (November 2012).

[50] YANG, C., HARKREADER, R. C., AND GU, G. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *Proc. of RAID* (2011).

[51] YANG, Y., AND PEDERSEN, J. O. A comparative study on feature selection in text categorization. In *Proc. of ICML* (1997).

[52] YANG, Z., WILSON, C., WANG, X., GAO, T., ZHAO, B. Y., AND DAI, Y. Uncovering social network sybils in the wild. In *IMC* (2011).

[53] ZHANG, L., YANG, J., AND TSENG, B. Online modeling of proactive moderation system for auction fraud detection. In *Proc. of WWW* (2012).

[54] ZHANG, Y., GE, Z., GREENBERG, A., AND ROUGHAN, M. Network anomography. In *Proc. of IMC* (2005).

DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse

Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS
Purdue University, West Lafayette, IN 47907
{*bsaltafo, gu16, xyzhang, dxu*}*@cs.purdue.edu*

Abstract

State-of-the-art memory forensics involves signature-based scanning of memory images to uncover data structure instances of interest to investigators. A largely unaddressed challenge is that investigators may not be able to *interpret the content of data structure fields*, even with a deep understanding of the data structure's syntax and semantics. This is very common for data structures with application-specific encoding, such as those representing images, figures, passwords, and formatted file contents. For example, an investigator may know that a `buffer` field is holding a photo image, but still cannot display (and hence understand) the image. We call this the *data structure content reverse engineering* challenge. In this paper, we present DSCRETE, a system that enables automatic interpretation and rendering of in-memory data structure contents. DSCRETE is based on the observation that the application in which a data structure is defined usually contains interpretation and rendering logic to generate human-understandable output for that data structure. Hence DSCRETE aims to *identify and reuse* such logic in the program's *binary* and create a "scanner+renderer" tool for scanning and rendering instances of the data structure in a memory image. Different from signature-based approaches, DSCRETE avoids reverse engineering data structure signatures. Our evaluation with a wide range of real-world application binaries shows that DSCRETE is able to recover a variety of application data — e.g., images, figures, screenshots, user accounts, and formatted files and messages — with high accuracy. The raw contents of such data would otherwise be unfathomable to human investigators.

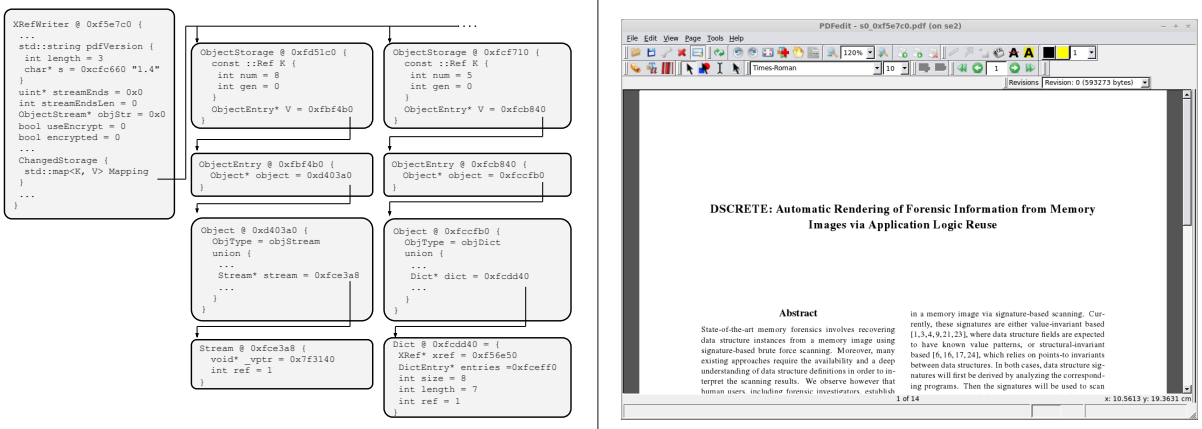
1 Introduction

Traditionally, digital investigations have aimed to recover evidence of a cyber-crime or perform incident response via analysis of non-volatile storage. This routine involves powering down a workstation, preserving images of any storage devices (e.g., hard disks, thumb drive, etc.), and later analyzing those images to recover evidentiary files. However, this procedure results in a significant loss of *live evidence* stored in the system's RAM — executing processes, open network connections, volatile IPC data, and OS and application data structures.

Increasingly, forensic investigators are looking to access the wealth of actionable evidence stored in a system's memory. Typically, this requires that an investigator have access to a suspected machine, prior to it being powered down, to capture an image of its volatile memory. Further, memory acquisition (both hardware [6] and software [25] based) must be as minimally invasive as possible since they operate directly on the machine under investigation. The resulting memory image is then analyzed offline using memory analysis tools. Therefore, the goal of memory analysis tools (like the work presented in this paper) is to recreate, in the forensics lab, the system's previously observable state based on the memory image.

Uncovering evidence from memory images is now an essential capability in modern computer forensics. Most state-of-the-art solutions locate data structure instances in a memory image via signature-based scanning. Currently these signatures are either value-invariant based [2, 3, 9, 21, 23, 26], where data structure fields are expected to have known value patterns, or structural-invariant based [5, 16, 17, 24], which rely on points-to invariants between data structures. In both cases, data structure signatures will first be derived by analyzing the corresponding programs. Then the signatures will be used to scan memory images and identify instances of the data structures. Contents of the identified instances will be presented to forensic investigators as potential evidence.

A significant challenge, not addressed in existing memory forensics techniques, is that investigators may not be able to *interpret the content of data structure fields*, even with the data structure's syntax and seman-



(a) Signature-based scanner output.

(b) DSCRETE-based scanner output.

Figure 1: Illustration of content reverse engineering challenge. (a) Raw content of an in-memory data structure instance representing a PDF file. (b) The same data structure after applying DSCRETE's scanner based on content reverse engineering.

tics. This is very common for data structures with application-specific encoding, such as those representing images, passwords, messages, or formatted file contents (e.g., PDF), all of which are potential evidence in a forensic investigation. For example, an investigator may know that a `buffer` field is holding a photo image (through existing data structure reverse engineering and scanning techniques [9, 15–17, 24, 26]), but still cannot display (and hence understand) the image. Similarly, a `message_body` field may hold an instant message, but the message is encoded, and hence it cannot be readily interpreted. We call this the *data structure content reverse engineering* challenge.

To enable automatic data structure content reverse engineering, we present DSCRETE[1], a system that automatically *interprets and renders* contents of in-memory data structures. DSCRETE is based on the following observation: The application, in which a data structure is defined, usually contains interpretation and rendering logic to generate human-understandable output for that data structure. Hence the key idea behind DSCRETE is to *identify and reuse* such interpretation and rendering logic in a binary program — without source code — to create a "scanner+renderer" tool. This tool can then identify instances of the data structure in a memory image and, most importantly, render them in the application's *original output format* to facilitate human perception and avoid the overhead of reverse engineering data structure signatures required by signature-based memory image scanners.

To illustrate the challenge of data structure content reverse engineering, we present a concrete example (from Section 4). Figure 1a shows the raw content of an in-memory data structure graph representing a PDF file. This is the output produced by existing signature-based scanners. For comparison, Figure 1b shows the same data structure content *after* applying DSCRETE's scanner with content reverse engineering capability. It is quite obvious that the reverse-engineered content would be far more helpful to investigators than the raw data structure content.

We have performed extensive experimentation with DSCRETE using a wide range of real-world commodity application binaries. Our results show that DSCRETE is able to recover a variety of application data — e.g., images, figures, screenshots, user accounts, and formatted files and messages — with very high accuracy. The raw contents of such data would otherwise be unfathomable to human investigators.

The remainder of this paper is organized as follows: Section 2 presents an overview of DSCRETE. Section 3 details the design of DSCRETE. Section 4 presents our evaluation results. Section 5 discusses some observations from our evaluation, current limitations, and future directions for this research. Section 6 discusses related works and Section 7 concludes the paper.

2 System Overview

2.1 Key Idea: Executable Code Reuse

DSCRETE is based on reusing the existing data structure interpretation and rendering logic in the original application binary. As a simple example, consider the Linux `gnome-paint` application. At the high-level,

[1]DSCRETE stands for "Data Structure Content Reverse Engineering via execuTable rEuse," pronounced as "discrete."

`gnome-paint` works as follows: An input image file is processed into various internal application data structures. The user then performs edits to and saves the image. To save the image, `gnome-paint` will reconstruct a formatted image from its internal data structures and write this image to the output file.

Later, if a forensic investigator wanted to recover the edited image left by `gnome-paint` in a memory snapshot, DSCRETE would be used to identify and automatically reuse `gnome-paint`'s own data structure rendering logic. First, DSCRETE will identify and isolate the corresponding data structure printing functionality within the application binary. For brevity, let us refer to this printing/rendering component as the function *P*. *P* should take as input a data structure instance and produce the human readable application output which is expected for the given data structure. In the case of `gnome-paint`, this component is the `file_save` function. It takes as input a `GdkPixbuf` structure and outputs a formatted image to a file. Note that *P* may not be a function in the programming language sense, but instead a *subsection of the application's code* responsible for converting instances of a certain data structure into some human-understandable form (e.g., output to the screen, file, socket, etc.).

Once *P* is identified, DSCRETE will reuse this function to create a *memory scanner+renderer* (or "scanner" for short) to identify all instances of the subject data structure in a memory image. If *P* is well defined for the input data structure, then one can expect *P* to behave erroneously when given input which is not a valid instance of that data structure. Under this assumption, we can present each possible location in the memory image to *P* and see if *P* renders valid output for the data structure of interest. We note that should an investigator alternatively choose to use a signature-based memory scanner to locate data structure instances, the DSCRETE-generated scanner *is still able to render* any located instances.

2.2 Overview of DSCRETE Workflow

Figure 2 presents the key phases and operations of DSCRETE. The first input is a binary application for which an investigator wishes to recover application data of interest from a memory image. To avoid compatibility issues (such as different data structure field layouts), this binary should be the same as the one that has contributed to the memory image.

The subject binary is then executed under instrumentation to identify the code region responsible for converting a specific data structure into application output (the function *P* defined earlier). We refer to this phase of DSCRETE as "tracing," and the details of this step are presented in Section 3.1. In the next phase, "identification" (Section 3.2), a *graph closure algorithm* is used to formulate a list of possible candidates for *P*. Each candidate is tested, by the "tester" component (Section 3.3), with a ground truth data structure instance to determine if it can serve as a viable memory scanner.

Once the specific application logic (*P*) is identified, DSCRETE packages this logic as a *context-free memory scanner* (Section 3.4), which will be presented to forensic investigators to scan and interpret memory images in this and future investigations involving the same application. We point out that the first three phases (tracing, identification, and tester) are a one-time procedure internal to DSCRETE and do *not* involve field investigators who will be using the DSCRETE scanners in their practice.

It is important to note that, unlike signature-based memory scanning techniques, we do not attempt to find and return the raw contents (bytes) of identified data structure instances in a memory image. Instead, we aim to present the investigator a set of *application-defined outputs* that would naturally be generated by the subject application, had it executed *P* with the data identified in the memory image. We emphasize that DSCRETE does *not* infer data structure definitions (unlike [17, 24]), nor does it derive data structure signatures (unlike [16]).

2.3 Assumptions and Setup

Firstly, we assume that when producing DSCRETE-based memory scanners (which is typically the task of a central lab of a law enforcement agency), the subject binary can be executed. This includes recreating any execution environment (i.e., operating system and application version, required libraries, directory configurations, etc.) which the application requires. We believe that this assumption is not overly difficult to realize. In a real forensic investigation, such runtime configuration information can be collected via preliminary examination of suspect or victim machines. Additionally, our dynamic instrumentation requires that address space layout randomization (ASLR) be turned off during the production of the DSCRETE memory scanners (i.e., only the investigator's personal workstation, *not* the suspect machine under study). The reason for this will become clear in Section 3.3.

Secondly, we assume that the OS kernel's paging data structures in the subject memory image are intact. This is a similar assumption made by many previous memory forensics projects [16, 21, 26]. We require this because DSCRETE takes as input only the subject application's memory session from the suspect machine. For our evaluation, we extracted the memory pages directly from running applications — which is preferred when an investigator has physical access to a suspect's ma-

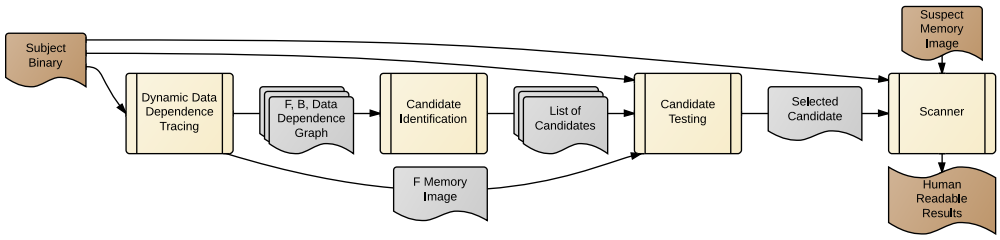


Figure 2: Overview of DSCRETE workflow.

chine. However in many forensic investigations only the memory snapshot and hard disk image are available. In this case the Volatility [26] `linux_proc_maps` and `linux_dump_map` plug-ins (or `memmap` and `memdump` for Windows) can be used to identify and extract process pages and mapping information from a whole system memory image.

3 System Design

In this section, we explain each phase of DSCRETE.

3.1 Dynamic Data Dependence Tracing

The first phase of DSCRETE, "tracing," collects a dynamic data dependence trace from the subject application binary. This trace must contain some portion of the future scanner's code (i.e., the code responsible for rendering a data structure of interest as human-understandable output). To collect this trace, we (as the central lab staff producing the scanners for field investigators) interact with the application to perform the following actions:

1) Create and populate an instance of the data structure used to store the data of interest. However, we make no assumptions on the knowledge about this data structure. We only assume that *some* data structure exists in the application which holds forensically interesting information in its fields.

2) The data structure of interest must be emitted as observable outputs (e.g., to files, network packets, or displayed on screen). This is to allow the scanner production staff to express their forensic interest by marking (part of) the output.

Again let us use `gnome-paint` to illustrate this procedure. To accomplish Step 1, we only need to execute `gnome-paint` with some input image. This will cause `gnome-paint` to create and populate numerous internal data structures to store the image's content. To accomplish Step 2, we only need to save the image to an output file. `gnome-paint` will process the image for output and call the GDK library's `gdk_pixbuf_save` function with the image's content as a parameter. While this may seem like a highly simplified example, the case studies in Section 4 show that in general we do not need to perform lengthy or in-depth interaction with an application to accomplish these two requirements.

Meanwhile, DSCRETE will be collecting each instruction's data dependence and recording any library functions or system calls invoked by the application as well as their input parameters. This is used to later identify which known external functions, specifically those which emit data to external devices, were invoked with the forensically interesting content as a parameter (`gdk_pixbuf_save` from our `gnome-paint` example). Note that since our analysis is at the binary level without symbolic information, we consider a parameter to be any memory read inside a function that depends on a value defined prior to the function's invocation. The memory may be accessed inside the function, subsequent functions, or as an argument to a system call[2], and the content read is logged as parameters. We exclude any memory not previously written to by the application or a previous library function, allowing us to ignore any memory which is private to the library function and not related to the parameter (i.e., the transformed data structure). This logging results in an output file containing the list of invoked external functions and parameters to each (similar to the Linux `strace` utility).

It is important to note here that DSCRETE saves a snapshot of the process's stack and heap memory at the invocation of any external library function which leads to an output-specific system call (i.e., `sys_write`, `sys_writev`, etc.). We (as the forensics lab staff) may, optionally, further specify individual library functions for which a snapshot should be taken. For example, if we know that the forensic evidence will be rendered on the application's GUI, then we may choose to only log visual-output related library calls in the GTK library. These snapshots will later be used to test possible *closure points* (defined in Section 3.2).

Once Steps 1 and 2 are accomplished, we may terminate the subject binary and search the log of external function calls for one in which the forensically in-

[2]We assume that system call interfaces are known and thus we can mark which parameters and memory ranges are read and which are written to.

teresting data is seen as parameters. Once suitable functions are chosen, DSCRETE only needs to identify which bytes of the parameters for those function invocations are of forensic value.

The chosen function invocations and set of parameter bytes will be important for two reasons: First, the parameter bytes will serve as the source nodes in our data dependence graph. Second, the function(s) will be used as the termination point for our scanner and the corresponding bytes will be the *output* of the scanner. For brevity, these functions will be referred to as F and the selected forensically interesting bytes of F's parameters as the set B. For our running `gnome-paint` example, consider `gdk_pixbuf_save` as F and the image buffer it prints to the output file as B.

Finally, a data dependence graph is generated using the trace gathered during dynamic instrumentation. The graph begins with the instructions responsible for computing the bytes of B as source nodes. Then in an iterative backwards fashion, any instruction which a graph node depends on is also added to the graph. Eventually, the graph will contain any instruction instance which led to the final value of B's bytes. This process is identical to that of typical dynamic slicing [13], we just chose to ignore control dependence as it is not required for identifying the functional closure (to be described next).

3.2 Identifying Functional Closure

Given F, B, and the data dependence graph, DSCRETE must find a *closure point* for the rendering function P. We define a closure point as an instruction in the data dependence graph which satisfies: 1) It directly handles a pointer to the forensically interesting data structure and 2) Any future instruction which reads a field of the data structure must be dependent on the closure point. This leads to the nice property that by changing the pointer handled by the closure point, we can change the data output by P. Returning to the `gnome-paint` example, the closure point is the instruction which moves a `GdkPixbuf` pointer into an argument register during `file_save`.

However, without source code or the effort of reverse engineering the subject binary, we cannot know the closure point for certain a priori. In fact, there may be multiple closure points in a program, any of which will satisfy our criteria above. To find a valid, usable closure point we use a combination of a graph closure algorithm and heuristics to output a list of *closure point candidates*. Each candidate is a tuple of the following: the address of an instruction which may satisfy the above criteria, the register or memory operand which it stores a pointer to, and the value of that pointer from the data dependence trace taken during tracing (Section 3.1).

Algorithm 1 Identifying *Closure Point* Candidates

```
Input: DataDepGraph(V, E), p
Output: Candidates[]
  SubGraphs[] ← ∅
  Previous_Candidate ← ∅
  for node n ∈ V in Reverse Temporal Order do
    G(Vn, En) ← ∅                          ▷ Build subgraph rooted at n
    Vn ← {n}
    for (n, t) ∈ E do          ▷ Each t that depends on n (may be ∅)
      Gt(Vt, Et) ← SubGraphs[t]            ▷ SubGraph rooted at t
      Vn ← Vn ∪ Vt
      En ← En ∪ Et ∪ (n,t)
    SubGraphs[n] ← G
    if Is_Store_Instruction(n) then        ▷ Apply heuristics to n
      val ← Stored_Value(n)
      loc ← Store_Location(n)
      if Is_Possible_Pointer(val) then
        if |SubGraph[n]|>|SubGraph[Previous_Candidate]| then
          Candidates ← Candidates ∪ (n, loc, val)
          Previous_Candidate ← n
    if |SubGraphs|> p%×|DataDepGraph| then
      break                    ▷ Only consider p% of DataDepGraph
```

We call this phase "candidate identification." The algorithm to identify closure point candidates is given in Algorithm 1. Starting from each byte in B, the algorithm steps through the data dependence graph in reverse temporal order (i.e., from the last instructions executed to the first). For each node visited (n) the algorithm builds a graph containing all previously visited nodes which depend on n (G in Algorithm 1). Essentially, graph G will resemble a subgraph rooted at n with its leaves accessing some bytes of B.

For each node n added to these subgraphs, the algorithm performs the following heuristic checks; any node which passes these checks is considered a closure point candidate. First, n must store a value (either to a register or memory location) which could be a possible data structure pointer (any integer value that falls within a memory segment marked readable and writable). Second, the size of the dependence subgraph rooted at n must be larger than the previous candidate's subgraph. The intuition here is that a correct closure point will take as input a pointer to a data structure instance, and store this pointer to be reused by the rendering function P. Thus for the part of the data dependence graph responsible for rendering a data structure instance, the largest subgraph must have the closure point at its root. Consider a data dependence graph for the `file_save` function from `gnome-paint`: The largest subgraph of this data dependence graph should be rooted at the input `GdkPixbuf` pointer.

Another heuristic is to stop the algorithm after only a small percent of the data dependence graph is analyzed. Note that the data dependence graph contains instructions from F back to the application's `main` function. Further, P will be close to F in the graph and signif-

icantly smaller than the rest of the application's code. This percentage is taken as a configurable input (p in Algorithm 1) and is set via a forward iterative approach. In our evaluations in Section 4, we started with a p value of 1 and incremented p until a valid closure point was found. Even in the extreme case (`top`), p was never more than 10 and was often less than 5.

In all of our evaluations, the number of candidates never exceeded 102 and was often below 30. Additionally, as will be explained in the next section, we never need to verify (or even see) any of the candidates. The testing of candidates is done mostly automatically.

3.3 Finding the Scanner's Entry Point

To test each closure point candidate, DSCRETE will run a modified version of the memory scanner described in the next section. This modified scanner, named the candidate "tester," takes as input: 1) the known end point of the scanner (i.e., F), 2) the memory image taken when F was executed, 3) the list of candidates, and 4) the subject binary. The modified scanner will treat F's memory image as the "suspect" memory image to scan. We assume that this memory image contains a valid instance of the data structure which held the data seen in B because the application was in the process of rendering/emitting this data structure instance's fields when the memory image was captured.

The candidate tester will re-execute the subject binary from the beginning, but before the process is started the scanner maps the "suspect" memory image's segments into the address space. Each segment (a set of pages) is mapped back to the address from which it was originally taken[3]. This ensures that pointers in these memory segments will still be valid in the new process's address space. Note that ASLR is disabled during DSCRETE operations. At this point, the new process is unaware of the added memory segments and executes normally using only its new allocations. Later, we will intentionally force the new process to use a small portion of the old process's data session to test closure point candidates, a technique we call *cross-state execution* (discussed in Section 3.5).

In the new execution, the forensically interesting data seen in this run of the application should be altered (e.g., executing `gnome-paint` with a different image). This will later allow the user to easily determine which candidate's output is correct (from the data structure in the memory image).

[3]We have not seen any cases where critical segments overlapped. This is because the segments are being mapped into ranges usually reserved for heap and stack space. Since these segments are almost universally relocatable the new process is simply allocated pages around our memory image.

The application runs until a closure point candidate instruction is executed. Here, the tester forks an identical copy-on-write child of the subject application to perform the actual scan; the parent process will be paused until the child has completed. The scanner looks up which register or memory operand this candidate stores its pointer value into and overwrites this location with the pointer's value stored in the candidate. Note that if this candidate is a correct closure point, then the stored pointer value is a valid pointer to a data structure instance in the mapped memory image. This assumes that the data structure instance is not corrupted from the beginning of the rendering function P (for which this candidate may be an entry point) to the invocation of F. Since all candidates are reasonably close to the invocation of F (within p% of the total trace size), we find that this is never a problem in practice.

Further, if this candidate is a correct closure point, the child process will now execute P, access the old process's memory segments (via the changed pointer value), generate the same bytes for B, and invoke F with these bytes. Imagine that, for our `gnome-paint` example, this candidate is the instruction which moves a `GdkPixbuf` pointer into a register during `file_save` (P). Now `file_save` will execute in the child process with the `GdkPixbuf` structure inside the memory image and should call `gdk_pixbuf_save` (F) with an identical image as was previously rendered (B). Also, recall that the forensically interesting information seen in the new run is altered. This is to easily partition between output generated from the memory image and output from the new execution of the application.

During testing, if the child process crashes after the pointer replacement, then the candidate is assumed incorrect and thrown out. When the F function(s) execute to completion (recall that in our `gnome-paint` example F is `gdk_pixbuf_save`) then the content given as input to F is recorded as a result for this closure point candidate test. An example of this recorded output is given later in Figure 6 (Section 4.3.1). The end of a scan is determined as follows: When F is a single function invocation, the child process is killed after F returns. If F consists of multiple invocations, the scan continues until the execution call stack returns to a point before the closure point. The parent process is then resumed, and this is repeated until all candidates are tested. A candidate is considered a valid closure point if it has accurately recreated the bytes chosen for B.

3.4 Memory Image Scanning

Once the data structure rendering function P has been identified, DSCRETE can build a memory scanning+rendering tool out of the subject binary. In fact, the production memory scanner is quite similar to the mod-

ified scanner used for testing candidates in the previous section. The difference is that we do not know where in a suspect memory image the data structures may be. The input to the memory image scanner tool are: 1) the chosen entry point and exit point of the printing function *P*, 2) the subject binary, and 3) the suspect memory image (as described in Section 2.3).

Again the scanner will re-execute the subject binary with the suspect memory segments mapped back into their original placements. Like before, the suspect memory segments will not be used until scanning begins, and until then the process executes using only its new allocations. With the same application input from candidate testing, the execution will reach *P*'s entry point, where the scanner pauses the application. For each address in the memory image, the scanner will fork an identical copy-on-write child and assign *P*'s pointer to the next address in the memory image. In essence, the scanner is executing *P* with a pointer to each byte of the suspect memory image. The scanning child process executes until *P*'s end point (as defined in the previous section) and then *P*'s output is recorded to a log or the child process crashes.

The intuition behind re-executing the application from the beginning is to automatically rebuild any dependencies required by *P*. DSCRETE requires that *P*'s only input be a pointer to a possible data structure. In reality, *P* may depend on multiple parameters set up by the application prior to the closure point. By re-executing the application from the beginning, we ensure that any other dependencies *P* has are taken care of before the scanner injects a data structure pointer.

The execution of *P* is done in a child process to isolate side effects. Not surprisingly, the vast majority of addresses will cause invalid memory accesses or other exceptions, and by scanning each byte in a separate process the scanner ensures that side effects do not contaminate future scans or global values. To speed up scanning, multiple child processes can be spawned to run in parallel.

In some rare cases, *P* is too simple (performs too little input processing) to crash on invalid input. For such cases, we allow for a user-defined post-processing phase. We still assume no use of source code or reverse engineering effort, but the user may perform sanity checks based on the known format or value ranges for an application's output. For instance, in our `top` case-study we had to remove any output which had a negative process ID or blank user or process name field. In our experiments, only three cases — `CenterIM`, `top`, and `Firefox VdbeOp` — required any post-processing. Further, this only occurs for very simple textual *P* functions — complex cases such as those requiring content reverse engineering naturally involve more strict parsing and input sanitization.

3.5 Cross-State Execution

DSCRETE maps one process's address space into the address space of another. Further, when DSCRETE executes the function *P*, this code will evaluate data in both the old and new address spaces. Once DSCRETE replaces a data structure pointer at the closure point, the scanning process will then access fields from the data structure in the old address space while still using stack and other heap objects in the new address space.

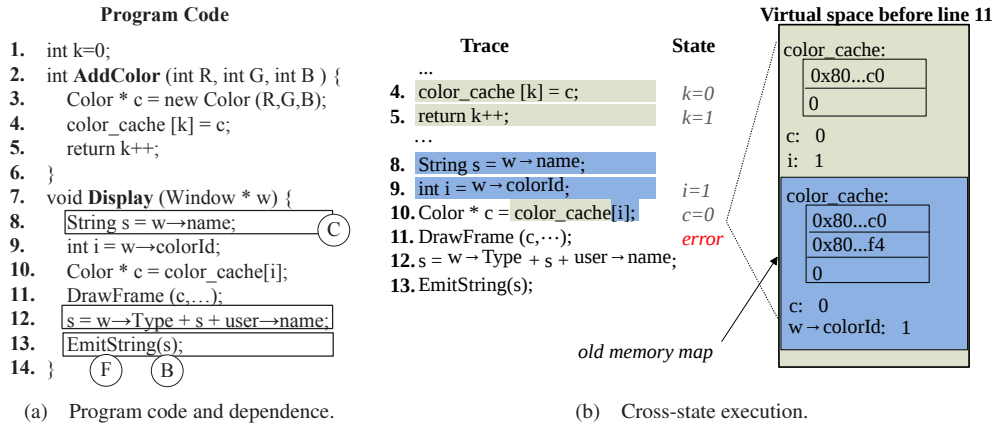
Ideally, any sub-execution that depends on the closure point would exclusively access the state from the old address space. In other words, we expect the continuation after the pointer replacement would consist of two disjoint sub-executions, one corresponding to running *P* on the old address space and the other corresponding to the rest of the execution exclusively on the new address space. However, due to the complex semantics of real world programs, such separation may not be achievable. There are two possible problems: 1) An instruction execution may depend on state from both address spaces, resulting in some state that is infeasible in either the original or the new execution. We call such instructions *confounded instructions*. 2) Since the old memory snapshot may not be complete, an instruction may access a location in the old space that is not mapped in the new space. Note that this location may now correspond to a valid address in the new space such that the access becomes one to the new space. We call this a *trespassing instruction*. Both could cause crashes and hence false negatives.

Consider the example in Figure 3. Figure 3a shows two functions[4]. The first function (lines 2 - 6) creates a `Color` object and adds it to the `color_cache`. The other (lines 7 - 14) renders a window, including drawing the `Color` to a frame and emitting the window title as a string. Note that different executions may add different `Color` objects to the cache. Specifically, the number of `Color` objects and their order vary across executions. Later, the window rendering function will look up a `Color` object from the cache using its id.

Assume we (as forensics lab staff) mark the `EmitString()` function at line 13 as *F* and `s` (the window title) as *B*. Following the candidate identification algorithm, we compute the backward data-dependence of *B* as those boxed statements. We further identify line 8 as the closure point candidate.

However, when we test this candidate, cross-state execution leads to undesirable results if not properly handled. Let us assume that two `Color` objects were cached during the original execution, whereas only one `Color` is added in the candidate test execution. Figure 3b shows the trace of the candidate test execution on the left, and,

[4]Our discussion is at the source code level for readability, whereas our design and implementation assume only the application binary.



(a) Program code and dependence.

(b) Cross-state execution.

Figure 3: Example for cross-state execution.

on the right, it shows the state of the new address space right before the execution of line 11. Note that the pages of the old address space are mapped inside the new address space. Each executed statement in the trace is colored based on the address space it operates on. Particularly, lines 4 and 5 execute before the pointer `w` is replaced at line 8, and hence belong to the new space. In contrast, lines 8 and 9 belong to the old space, as their values are loaded from locations derived from the replaced `w`. Line 10 is a confounded instruction, as the array `color_cache` belongs to the new space while `i` belongs to the old space. As a result, an invalid color is loaded, leading to a crash. However, observe that lines 10 and 11 are not in the data dependence of B, as such we could potentially skip them.

Therefore, given a closure point candidate C and its termination point F, DSCRETE scans the original execution trace from C to F during the candidate identification phase. For each address dereference it encounters, it tests if the address is exclusively dependent on the pointer parameter at C. If not, it is a confounded dereference. DSCRETE further tests if the dereference is in the data-dependence graph of B, and if not, marks the instruction as an irrelevant dereference to be skipped during test execution and later scanning executions. In practice, we observed confounded memory dereferences in only one of the cases we studied.

Handling trespassing instructions is relatively easier. Given a closure point candidate C and its termination point F, DSCRETE scans the original execution trace from C to F and marks each address dereference that it encounters and is dependent on the pointer parameter at C. At runtime, if a marked dereference accesses a location in the new space, it is a trespassing access and can be skipped.

4 Evaluation

DSCRETE leverages the PIN binary analysis platform [19] to perform instrumentation. Since PIN executes before the subject binary is loaded, this allows us to map the memory image into the new process's address space before the operating system's loader can claim stack and heap regions. DSCRETE relies on minimal OS-specific knowledge (i.e., system call and application binary interface definitions), thus DSCRETE can easily be ported to any operating system that PIN supports. In the remainder of this section, we present results from evaluating DSCRETE with a number of real-world applications and focus on a subset which highlight the use of DSCRETE and a few critical observations.

4.1 Experimental Setup

Our evaluation used a Ubuntu 12.10 Desktop system as the "suspect" machine. Each application was installed on the machine and interacted with by the authors to generate sufficient allocations and deallocations of data structures. We used `gdb` to capture memory images from the application periodically during the system's use. To attain ground truth, we manually instrumented the applications to log allocations and deallocations for data structures corresponding to the output of forensic interest (i.e., B in Section 3.1). This log was later processed to measure false positives (FP) and false negatives (FN). For analysis, we employed a Ubuntu 12.10 virtual machine. To recreate the suspect machine's running environment, we copied the applications and needed configuration files from the suspect machine's hard disk. We then performed all forensic investigation within the virtual machine.

Application	F	Forensically Interesting Data	Size B (bytes)	p%	#C	#O	#P
CenterIM	`SSL_write`	Username & Password	336	5%	46	1	1
convert	`fwrite`	Output Image Content	81902	9%	18	7	2
gnome-paint	`gdk_pixbuf_save`	Image Content	670900	1%	18	2	2
gnome-screenshot	`gdk_pixbuf_save_to_stream`	Screenshot Content	1139791	1%	5	4	3
gThumb	`gtk_window_set_title`	File Info Window Title	85	1%	102	4	2
	`gdk_pixbuf_save_to_bufferv`	Image File Content	20360	1%	10	3	3
Nginx	`write`	HTTP Access Log	181	5%	25	1	1
PDFedit	`fwrite, fputc`	Edited PDF Content	30416	1%	46	6	3
SQLite3 Shell	`fputs`	Database Query Results	19	2%	4	1	1
	`fprintf`	Database Op. Log	38	2%	17	5	1
top	`putp`	Process Data	132	10%	1	1	1
Xfig	`fprintf`	Figure Content	1001	1%	9	3	3

Table 1: Results from identifying applications' P functions (#C shows the number of identified candidates, #O shows how many of those produced output, and #P shows the final subset which are valid closure points).

4.2 Function Identification Effectiveness

This section presents results of isolating the data rendering function P in each tested application. From the `CenterIM` instant messenger, we target the component which emits the user's login and password (still in plain text) to an SSL socket. Also, given the importance of image content to investigations, we isolate image rendering functions from three common image editors: `convert`, `gnome-paint`, `gThumb`, as well as the `gThumb` GUI function which displays the current image's name to the window title. The output function of `gnome-screenshot` can allow an investigator to see what screen-shot a suspect was capturing. Additionally, we reuse `Xfig`'s figure saving P function to reconstruct a vector figure that was being worked on. As we introduced in Section 1, the PDF saving functionality of `PDFedit` allows investigators to recover the edited PDF file. For internal application data, we identified P functions for `SQLite`'s query results and operations log (more on how these scanner+render tools are used later in this section). It is very common for attackers to tamper with server log files, so we isolated the `Nginx` webserver's connection logging function, thus an investigator can compare with the uncovered in-memory connections. Finally, for details on all running processes in a suspect system, we identified the process data printing logic in the `top` utility.

Table 1 shows a summary of the results from each of these applications. The application name and F function are shown in Columns 1 and 2 respectively. Column 3 details the forensically interesting data that were to be emitted by $F(B)$ and Column 4 shows the size of B in bytes. The percentage of the data dependence graphs used to generate candidates is shown in Column 5. Finally Columns 6 to 8 show the number of candidates identified by our algorithm (#C), how many of those produced any output (#O), and the final subset which accurately recreated B and could be used for valid closure points (#P), respectively.

From Table 1 we make the following observations: First, our algorithm/heuristics used to identify closure point candidates are accurate enough to limit the number of candidates to a reasonable search space. Although candidates are tested automatically during the candidate tester's execution, we aim to minimize the number of candidates to test. From Table 1, we see that 11 out of the 12 applications have less than 50 candidates. The only application with more than 50, `gThumb`, has 102, and as we see in Row 5 of the table, they are drastically narrowed down by the candidate tester. Manual investigation revealed that `gThumb`'s larger number of candidates was due to extra data dependencies caused by another parameter to its F function (`gtk_window_set_title`).

The second observation we make is that, of the total number of candidates identified, very few will be true closure points. This is intuitive since there is only one true entry to the P function in the application. Third, since the number of candidates which produced valid output is so small, it is relatively simple for a DSCRETE user to identify which candidate accurately reproduced B.

On average, each candidate testing component rendered application output for only three closure point candidates. The maximum, `convert`, rendered only seven outputs during candidate testing. Further, *more than half* of the applications produced ideal candidates — all candidates that rendered output were valid candidates. For the other five applications, about 45% of candidates

which produced output accurately recreated the expected forensically interesting data (i.e., the new output matched that seen before). This shows that: 1) Visually inspecting candidate output is a reasonably quick and practical task and 2) DSCRETE can identify and validate closure point candidates with high accuracy.

Table 1 shows that it is not uncommon for multiple correct closure points to exist for a *P* function. Manual investigation revealed that this is caused by two program features: nested data structure pointers and register-to-stack spilling. In the nested data structure situation, if a data structure A has a pointer to structure B and *P* uses the B pointer within A, then either the A pointer or its internal B pointer may be valid closure points for *P*. For the register-to-stack spilling situation, a pointer to an input data structure is initially stored in a register, but when contention forces that register to be spilled onto the stack, either the initial register or its later stack-saved location may be used for closure points.

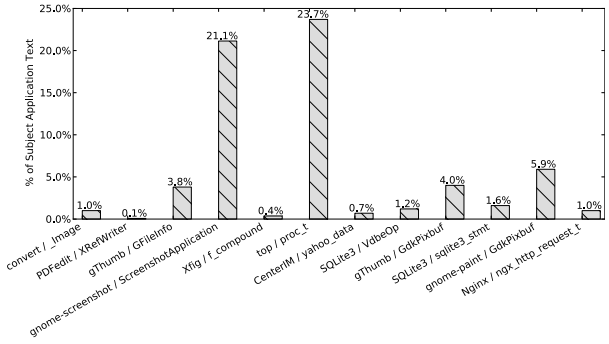


Figure 4: Normalized size of *P* vs. entire binary code.

Table 1 also shows that a valid closure point is typically located in the bottom 5% of the data dependence graph. Thus, the actual rendering function being reused is often only a small percentage of the binary's text. Figure 4 shows the normalized percentage of the host binary which we reuse for each scanning function. The size of the reused code is measured as the total in-memory size of all unique instructions observed during all re-executions of *P*. `Top`, `gnome-screenshot`, and `gnome-paint` are outliers due to the relatively small size of the applications and the resulting dependence graphs.

SQLite *P* Functions. An interesting application of DSCRETE can be seen in the experiments with SQLite. For these experiments DSCRETE was used with the SQLite3 command shell and a homemade database file to find *P* functions for a database query's result (`struct sqlite3_stmt`) and operations log (`struct VdbeOp`). These data structures are defined by the SQLite3 library and exported to client applications. The *P* functions DSCRETE identifies would be used to build memory scanner+renderer tools which could discover those data structures and render their content in the same format as the SQLite3 command shell.

These scanners could then be used on memory images from *any application which uses SQLite*. Since these data structures are defined by the SQLite library, any application using SQLite should transitively allocate and use these data structures. Further, we are reusing the SQLite3 command shell's *P* functions, so even if an application never outputs the data held in these structures, we can still discover and interpret them using the more general SQLite memory scanners. In the next section, we show results from applying these scanner+renderer tools to memory images from Mozilla Firefox and darktable image editor.

4.3 Memory Scanner Effectiveness

Table 2 reports the effectiveness of the DSCRETE-generated scanner+renderer tools when scanning a context-free memory image from each application. The application name is shown in Column 1. The subject data structure (input to *P*) and the structure's size are shown in Columns 2 and 3[5]. The number of true instances in the suspect memory image is shown in Column 4[6]. Column 5 shows the total number of output generated by each scanner+render tool. Columns 6 to 10 show the number of generated output which are: true positives (TP) - backed by true data structure instances, false positives (FP) and the percentage of FPs in the total output (FP%), and false negatives (FN) and the corresponding FN percentage.

This table shows that the *P* function identified by DSCRETE is almost always well defined. This allows DSCRETE to uncover and render valid data structure instances with 100% accuracy for most cases. Specifically, Table 2 shows that DSCRETE's scanner+renderer tools are perfectly accurate (i.e., no FP and no FN) in 11 out of the 13 cases. We analyze the two FP/FN cases in detail later in this section. More importantly, DSCRETE overcomes the data structure content reverse engineering challenge by displaying the results in each application's original output format. The test cases covered in Table 2 span a wide range of application data: usernames and passwords, images, PDF files, vector-based graphics, as well as formatted and unformatted textual output. This portrays the generality of DSCRETE and represents several key types of evidence that would be very difficult

[5]Such information was obtained via manual instrumentation, inspection, and reverse engineering only for the purpose of evaluation. DSCRETE does not need or have access to this information during operation.

[6]This includes all the data structure instances which were allocated and not yet released and overwritten when the memory image was captured.

Application	Subject Data Structure	Size (bytes)	True Instances	Total Output	TP	FP	FP%	FN	FN%
CenterIM	`yahoo_data`	160	1	1	1	0	0.0%	0	0.0%
convert	`_Image`	13208	1	1	1	0	0.0%	0	0.0%
darktable	`sqlite3_stmt`	272	1	1	1	0	0.0%	0	0.0%
Firefox	`sqlite3_stmt`	272	1	1	1	0	0.0%	0	0.0%
	`VdbeOp`	24	788	1384	753	502	40%	35	4%
gnome-paint	`GdkPixbuf`	80	51	51	51	0	0.0%	0	0.0%
gnome-screenshot	`ScreenshotApplication`	88	1	1	1	0	0.0%	0	0.0%
gThumb	`GFileInfo`	48	382	381	381	0	0.0%	1	0.4%
	`GdkPixbuf`	80	63	63	63	0	0.0%	0	0.0%
Nginx	`ngx_http_request_t`	1312	6	6	6	0	0.0%	0	0.0%
PDFedit	`XRefWriter`	344	1	1	1	0	0.0%	0	0.0%
top	`proc_t`	720	382	382	382	0	0.0%	0	0.0%
Xfig	`f_compound`	112	1	1	1	0	0.0%	0	0.0%

Table 2: Results from DSCRETE-generated scanner+renderer tools.

(if at all possible) to reconstruct from raw data structure contents.

Table 2 shows that many of the subject data structures are smaller than the resulting application output (*B* from Table 1). Our manual analysis of these structures reveals that 10 of the 12 data structures contain several pointers to other data structures used by *P*. This confirms our intuition that, in order to recover usable evidence from a memory image, numerous data structures must be uncovered and interpreted. Note that an investigator never actually sees any of these structures, but rather is presented only the application output rendered from the structures' contents. Figure 1a in Section 1 is one such example.

Another metric to report is the time taken to scan, which varies depending on: 1) the complexity of the rendering function *P* and 2) the size of the memory image being scanned. Figure 5 shows the scanning speed in bytes-per-second for each scanner function in our evaluation. During our experiments, the size of the applications' heaps ranged from 400KB to about 5MB, and total heap scanning time ranged between 15 minutes to just over 2 hours, with most taking about 30 to 45 minutes. Admittedly the scanning and rendering of evidence is slower than typical signature-based memory scanners, but still well within the typical time taken to process digital evidence, with the added benefit that evidence is presented in a human-understandable form. Ayers [1] points out that it may take "several hours or even days when processing average volumes of evidential data," which is confirmed by our collaborators in digital forensics practice.

False Positive and False Negative Analysis. We notice that only the gThumb and Firefox `VdbeOp` experiments experienced any negative results. Manual investigation into these two experiments' false negative results (i.e., true data structure instances not discovered by DSCRETE) revealed that those structures were allocated, but did not contain enough data to be rendered by *P*. They were either in the process of initialization or deletion or being used as empty templates by the application.

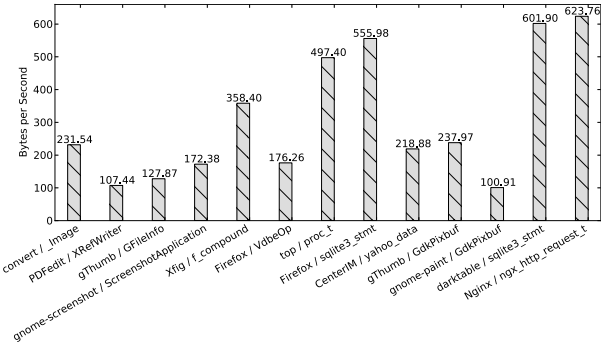


Figure 5: Observed throughput of each scanner.

Interestingly, the Firefox `VdbeOp` case study (SQLite's operations log structure) represents a counter-example to our hope that *P* be well defined. In this case, *P* performs little parsing and no sanity checks on its input. A `VdbeOp` structure is essentially a set of seven integer values, and SQLite3 uses these integers as indices in a global string table, without any sanity checks. Since this *P* function performs such trivial parsing, a large number of false inputs produce typical SQLite3 Shell output. We consider this a worst-case scenario for DSCRETE, and believe it is also the case for many other memory forensic techniques when facing such a trivial data structure.

In Section 1 we introduced one example of forensic

data which would be uninterpretable without data structure content reverse engineering. The complex multi-level data structure representing a PDF file requires non-trivial processing to locate the fields which contain any usable PDF content. Further, many fields are encoded, compressed, or computed only when outputting the PDF file. In the remainder of this section, we present several other application case studies with DSCRETE.

4.3.1 Case Study: `convert`

This case study highlights DSCRETE's content reverse engineering capability for image data structures. The `convert` utility is used to apply various transformations to an image file. The source image file is processed and converted into internal data structures, (i.e., an `_Image` and array of `_PixelPacket` structures). Various transformations (such as scaling, blurring, etc.) are applied, and the pixels are re-composed into an image and written to a file. It would be considerably difficult to reconstruct the image from its in-memory representation, even with a deep understanding of these structures' syntax and semantics. However, DSCRETE is able to overcome this challenge by identifying and reusing the image output component (function `WriteImage`) which constructs an output image file from an input `_Image` structure.

As shown in Row 2 of Table 1, *B* (the image's content) was seen as an argument to the `fwrite` function. Using this, DSCRETE identified 18 closure point candidates in the bottom 9% of the data dependence graph. Of these candidates, 16 clustered around the handling of `_PixelPacket` structures in the image reconstruction routine, and the remaining 2 candidates handled the input `_Image` structure at the entry to the `WriteImage` function.

The DSCRETE candidate tester component eliminated 16 candidates which handled `_PixelPacket` structures. For the remaining two candidates, DSCRETE produced the log and application output shown in Figure 6. From Figure 6 we see that Candidates 1 and 2 successfully executed *P* (ending with `fwrite`). More importantly, DSCRETE accurately rendered the `_Image` data structure's content – presenting proof that both candidates form valid *P* functions which can reconstruct the image seen previously. As Table 2 shows, this *P* function was well-defined and the resulting scanner located and rendered the "image of interest" in the memory image with no false positives or false negatives.

4.3.2 Case Study: `Xfig`

The second case study is with `Xfig`, in which data content reverse engineering is essential to uncovering usable evidence from data structure instances. `Xfig` is a Linux-based vector graphics editor which defines several types of data structures for different drawable shapes (i.e., ellipse, spline, etc.). From `Xfig`, we intended to build a scanner+renderer tool to reveal the figure a suspect was drawing. Referring back to Table 1, DSCRETE located 9 closure point candidates in the bottom 1% of the data dependence graph. DSCRETE tested these 9 candidates and decided that 3 of them which rendered output were valid closure points. One of those was chosen (DSCRETE prefers the closure point highest in the dependence graph) to build a scanner+renderer for `Xfig`'s `f_compound` data structure.

An `f_compound` structure is a container for several shape structures. Each shape structure stores its dimensions, coordinates, color, etc. In order to reconstruct a figure, each of these shape structures must be recovered from a memory image, interpreted, and shape-specific rendering functions must be invoked. Existing signature-based memory scanners could present an investigator with a list of shape data structures instances from a memory image, but without the interpretation logic and shape-specific rendering, the investigator cannot see what the figure looks like. By comparison, the DSCRETE-generated scanner+renderer can locate the figure's `f_compound` structure, traverse all the contained shape structures (in the *P* function), and output `Xfig`'s original figure content. Table 2 shows that this *P* function is well-defined and recovered the figure's content with 100% accuracy from the target memory image.

```
Candidate 1 ===== Scanning from 0x6a16c0:
  fwrite@libc ( 0x6ba360 ["<89>PNG<0d0a>"...], 1, 81902, 0x6b7320 [data] )
    Arg 1 written to file "c1_0x6ba360.out"

Candidate 2 ===== Scanning from 0x6a5c90:
  fwrite@libc ( 0x6ba360 ["<89>PNG<0d0a>"...], 1, 81902, 0x6b7320 [data] )
    Arg 1 written to file "c2_0x6ba360.out"
```

(a) Candidate test result log.

(b) Output image file for Candidate 1.

Figure 6: Candidate testing output. (a) Each *P* function is shown, similar to the Linux strace utility, with parameters seen during invocation. If the tester component is set for file output, the file name is also printed. (b) Shows the output file for Candidate 1.

4.3.3 What You Get Is More Than What You See

We observe that some applications will construct more data structures than they intend to display. Without content reverse engineering, these extra data structures would all need to be manually interpreted for investigation. DSCRETE intuitively renders such additional evidence, allowing an investigator to quickly determine if it is forensically valuable.

In our experiment with `top`, the true number of `proc_t` instances is 382, whereas while executing `top` only 31 processes were displayed at a time. Since all 382 `proc_t` structures were in `top`'s memory image, DSCRETE was able to uncover and present each as they would have been displayed by the original `top` process.

Another example is `gThumb`, which displays an image being edited and other images in the same directory. `gThumb`'s memory contained valid data structures for 63 images: 56 GUI icons and 7 suspect images, and DSCRETE recovered them all, including the 7 suspect images. More importantly, 3 of the 7 suspect images were not being displayed by the GUI. Without DSCRETE, determining which raw data structures were icons and which were evidence would require extensive manual effort. With DSCRETE, an investigator can immediately see the distinction. Note also, that those GUI icons are not false positives. Instead, they are valid and relevant image data structures, because the investigator may use such GUI artifacts to infer which application screen the suspect was focusing on.

5 Discussion and Future Work

Still at its early stage, DSCRETE represents a new approach for digital evidence collection. The prototype presented here has several limitations that will be addressed in our future work.

As mentioned in Section 3.5, cross-state execution may cause conflicting memory access patterns (i.e., confounded or trespassing instructions). DSCRETE selectively skips unnecessary instructions which may cause cross-state conflicts. However, this method is limited to the instructions recorded during tracing, and cannot reason about instructions that *were not executed*. Although we did not encounter such complications in our experiments, we do believe that they exist and will explore using static dependence analysis in the future.

DSCRETE relies on each application's own rendering logic to differentiate between valid and invalid input (data structures to be rendered). As we see in Section 3.4, this can be problematic if the rendering function performs very little input processing and validation. Our experiment suggests that this problem exists for highly simplified data structures, which may still be of forensic value. Handling such data structures is our ongoing work. Additionally, since DSCRETE reuses application binary logic, an interesting problem is to handle data which contains *exploits* against the rendering logic.

Another current limitation which we leave for future work is replacing multiple input data dependencies for a rendering function. Currently, DSCRETE identifies and replaces only a single data structure pointer seen as input to P. However, it is assumable that a single application output be generated from *multiple* unrelated data structures. Although we have not encountered such need, the problem is realistic and requires enhancements to the closure point identification and the scanning algorithms.

Like many binary analysis-based tools, DSCRETE is not yet ready to handle self-modifying code or binaries with highly obfuscated control flows, which may cause problems in dependence detection or state crossing. However, these problems are common in malware programs and hence worth solving. One future direction is to develop DSCRETE on an obfuscation-resistant binary analysis platform (e.g., [29]).

The methodology used in DSCRETE is designed to operate directly on a target machine binary. As such, it is not applicable to programs written in *interpreted* languages (e.g., Java). Such programming languages add layers of indirection between the machine instructions observed by DSCRETE and the application's true syntax and semantics (i.e., data structures and rendering functions). Developing new techniques to handle programs written in interpreted languages is an intriguing direction for our future research.

6 Related Work

Memory Image Forensics. Previous research in memory forensics has mainly centered around uncovering data structure instances using signature-based brute force scanning. Such techniques can be roughly classified into value-invariant based [2,3,9,21,23,26,27] and structural-invariant based [5, 15, 16].

Value-invariant signatures seek to classify data structures by the expected value(s) of their fields. More recently, DEC0DE [27] enhances value-based signatures with probabilistic finite state machines to recover evidence from smartphones. Structural-invariant based signatures are derived by mapping interconnected data structures. SigGraph [16] uses such signatures for brute-force memory image scanning. Later, DIMSUM [15] attempts to probabilistically locate data structure instances in un-mappable memory. Further, numerous forensic tools and reverse engineering systems [7, 14, 17, 20, 29] make use of data structure traversal.

Compared with these techniques, DSCRETE does not require data structure definitions or data structure field

value profiles as input. Moreover, DSCRETE can intuitively interpret data structure contents (e.g., rendering an image in memory). To the best of our knowledge, no existing memory forensic tool has similar capability.

Binary reverse engineering techniques [14, 17, 24] can reverse engineer data structure definitions (e.g., field types) from binaries. They can also reverse engineer semantic information to a certain extent. As such, they can be used in forensic analysis. However, these techniques can only reverse engineer semantics of generic data such as timestamps and IP addresses. Such approaches are hardly applicable to interpreting contents of application-specific and encoded data structure fields.

Binary Component Identification and Reuse. At the heart of the DSCRETE technique is application logic reuse. DSCRETE uses dynamic binary program tracing to identify which functional component of a binary application is responsible for generating forensically interesting output. It hence shares some common underlying techniques with existing binary identification and reuse techniques [4, 12, 18] and program feature identification [11, 28].

Similar to how DSCRETE employs a data dependence graph, Wong et. al. [28] use program slicing to identify the code region for a program feature. To further understand which application components contribute to an observed runtime behavior, Greevy et al. [11] use feature-driven dynamic analysis to isolate computational units of an application. In contrast, DSCRETE uses only an application's data dependence to identify candidates for later construction of a memory scanner+renderer.

Binary Code Reutilization (BCR) [4] involves using a combination of dynamic and static binary analysis to identify and extract malware encryption and decryption functions. The goal of BCR was to reuse such extracted logic as a functional component in a different program developed by the user. Inspector Gadget [12] uses dynamic slicing to identify specific malware behavior for extraction and later reuse/analysis. Lin et al. [18] suggested using dynamic slicing to identify applications' functional components to compose reuse-based trojan attacks. DSCRETE does not aim to extract application logic from a target binary, but rather re-execute it in-place to scan a memory image and render subject data structure contents.

Virtuoso [8] involves using dynamic slicing to identify logic from in-guest applications which could be reused for virtual machine introspection. However, Virtuoso is not able to handle input that is not encountered during off-line training. A DSCRETE-generated scanner can handle any input that the original *P* function could handle. Later, VMST [10] and Hybrid-Bridge [22] use system-wide instruction monitoring to allow introspection of one VM's kernel data from another. VMST redirects memory accesses for every instruction of the reused logic, whereas DSCRETE only needs to replace the data structure pointer at the closure point. Further, VMST relies on system call definitions to start logic reuse, while DSCRETE must automatically identify such a starting point (i.e., the closure point) in the subject binary.

7 Conclusion

We have presented DSCRETE, a data structure content reverse engineering technique which reuses application logic from a subject binary program to uncover and render forensically interesting data in a memory image. DSCRETE is able to recreate intuitive, human-observable application output from the memory image, without the burden of reverse engineering data structure definitions. Our experiments with DSCRETE show that this technique is able to effectively identify interpretation/rendering functions in a variety of real-world applications, and DSCRETE-generated scanner+renderer tools can uncover and render various types of data structure contents (e.g., images, figures, and formatted files and messages) from memory images with high accuracy.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. We are grateful for the suggestions and guidance from Dr. Golden G. Richard III. This research was supported, in part, by NSF under Award 1049303 and DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] AYERS, D. A second generation computer forensic analysis system. *Digital Investigation 6* (2009), 34–42.

[2] BETZ, C. Memparser forensics tool. `http://www.dfrws.org/2005/challenge/memparser.shtml`, 2005.

[3] BUGCHECK, C. Grepexec: Grepping executive objects from pool memory. In *Proc. 6th Annual Digital Forensic Research Workshop (DFRWS)* (2006).

[4] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proc. 17th Annual Network and Distributed System Security Symposium* (2010).

[5] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proc. 16th ACM Conference on Computer and Communications Security* (2009).

[6] CARRIER, B. D., AND GRAND, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1* (2004), 50–60.

[7] CASE, A., CRISTINA, A., MARZIALE, L., RICHARD, G. G., AND ROUSSEV, V. Face: Automated digital evidence discovery and correlation. *Digital Investigation 5* (2008), 65–75.

[8] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. 2011 IEEE Symposium on Security and Privacy* (2011).

[9] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *Proc. 16th ACM Conference on Computer and Communications Security* (2009).

[10] FU, Y., AND LIN, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proc. 2012 IEEE Symposium on Security and Privacy* (2012).

[11] GREEVY, O., AND DUCASSE, S. Correlating features and code using a compact two-sided trace analysis approach. In *Proc. 9th European Conference on Software Maintenance and Reengineering* (2005).

[12] KOLBITSCH, C., HOLZ, T., KRUEGEL, C., AND KIRDA, E. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proc. 2010 IEEE Symposium on Security and Privacy* (2010).

[13] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (1988), 155–163.

[14] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs. In *Proc. 18th Annual Network and Distributed System Security Symposium* (2011).

[15] LIN, Z., RHEE, J., WU, C., ZHANG, X., AND XU, D. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. 19th Annual Network and Distributed System Security Symposium* (2012).

[16] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. Sigraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. 18th Annual Network and Distributed System Security Symposium* (2011).

[17] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proc. 17th Annual Network and Distributed System Security Symposium* (2010).

[18] LIN, Z., ZHANG, X., AND XU, D. Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proc. 2010 IEEE/IFIP International Conference on Dependable Systems and Networks* (2010).

[19] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices* (2005), vol. 40.

[20] MOVALL, P., NELSON, W., AND WETZSTEIN, S. Linux physical memory analysis. In *Proc. USENIX Annual Technical Conference, FREENIX Track* (2005).

[21] PETRONI JR, N. L., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3* (2006), 197–210.

[22] SABERI, ALIREZA FU, Y., AND LIN, Z. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proc. 20th Annual Network and Distributed System Security Symposium* (2013).

[23] SCHUSTER, A. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation 3* (2006), 10–16.

[24] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. 18th Annual Network and Distributed System Security Symposium* (2011).

[25] SYLVE, J., CASE, A., MARZIALE, L., AND RICHARD, G. G. Acquisition and analysis of volatile memory from android devices. *Digital Investigation 8* (2012), 175–184.

[26] THE VOLATILITY FRAMEWORK. Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.

[27] WALLS, R., LEVINE, B. N., AND LEARNED-MILLER, E. G. Forensic triage for mobile phones with dec0de. In *Proc. USENIX Security Symposium* (2011).

[28] WONG, W. E., GOKHALE, S. S., AND HORGAN, J. R. Quantifying the closeness between program components and features. *Journal of Systems and Software 54*, 2 (2000), 87–98.

[29] ZENG, J., FU, Y., MILLER, K. A., LIN, Z., ZHANG, X., AND XU, D. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proc. 20th ACM Conference on Computer and Communications Security* (2013).

Cardinal Pill Testing of System Virtual Machines

Hao Shi
USC/Information Sciences Institute
haoshi@usc.edu

Abdulla Alwabel
USC/Information Sciences Institute
alwabel@usc.edu

Jelena Mirkovic
USC/Information Sciences Institute
mirkovic@isi.edu

Abstract

Malware analysis relies heavily on the use of virtual machines for functionality and safety. There are subtle differences in operation between virtual machines and physical machines. Contemporary malware checks for these differences to detect that it is being run in a virtual machine, and modifies its behavior to thwart being analyzed by the defenders. Existing approaches to uncover these differences use randomized testing, or malware analysis, and cannot guarantee completeness.

In this paper we propose Cardinal Pill Testing—a modification of Red Pill Testing [21] that aims to enumerate the differences between a given VM and a physical machine, through carefully designed tests. Cardinal Pill Testing finds five times more pills by running fifteen times fewer tests than Red Pill Testing. We further examine the causes of pills and find that, while the majority of them stem from the failure of virtual machines to follow CPU design specifications, a significant number stem from under-specification of the effects of certain instructions by the Intel manual. This leads to divergent implementations in different CPU and virtual machine architectures. Cardinal Pill Testing successfully enumerates differences that stem from the first cause, but only exhaustive testing or an understanding of implementation semantics can enumerate those that stem from the second cause. Finally, we sketch a method to hide pills from malware by systematically correcting their outputs in the virtual machine.

1 Introduction

In today's practice of analyzing malware [3, 14, 16, 26, 23], system virtual machines are widely used to facilitate fine-grained dissection of malware functionalities (e.g., Anubis [4], TEMU [6, 24], and Bochs [17]). For example, virtual machines can be used for dynamic taint analysis, OS-level information retrieval, and in-depth behavioral analysis. Use of virtual machines also protects the host, by isolating it from potentially malicious actions.

Malware authors have devised a variety of methods to hinder automated and manual analysis of their code, such as anti-dumping, anti-debugging, anti-virtualization, and anti-intercepting [10, 11]. Recent studies [7, 18] show that anti-virtualization and anti-debugging techniques have become the most popular methods of evading malware analysis. Chen et al. [8], find in 2008 that 2.7% and 39.9% of 6,222 malware samples exhibit anti-virtualization and anti-debugging behaviors respectively. In 2011, Lindorfer et al. [18] detect evasion behavior in 25.6% of 1,686 malicious binaries. In 2012, Branco et al. [7] analyze 4 million samples and observe that 81.4% of them exhibit anti-virtualization behavior and 43.21% exhibit anti-debugging behavior.

Upon detection of a virtual environment or the presence of debuggers, malicious code can alternate execution paths to appear benign, exit programs, crash systems, or even escape virtual machines. It is critically important to devise methods that handle anti-virtualization and anti-debugging, to support future malware analysis. In this paper we focus only on anti-virtualization handling, and specifically on CPU semantic attacks.

We observe that malware can differentiate between a physical and a virtual machine due to numerous subtle differences that arise from their implementations. Let us call the physical machine an *Oracle*. Malware samples execute sets of instructions with carefully chosen inputs (aka *pills*), and compare their outputs with the outputs that would be observed in an Oracle. Any difference leads to detection of VM presence.

These attacks are successful because there are many differences between VMs and physical machines, and existing research in VM detection [21, 20, 15] uses ad-hoc tests that cannot fully enumerate these differences. Since malware is run within a VM, all its actions are visible to the VM and all the responses are within a VM's control. If differences between a physical machine

and a VM could be enumerated, the VM could use this database to provide expected replies to malware queries, thus hiding its presence. This is akin to kernel root kit functionality, where the root kit hides its presence by intercepting instructions that seek to examine processes, files and network activity, and provides replies that an uncompromised system would produce.

In this paper we attempt to enumerate all the differences between a physical machine and a virtual machine *that stem from their differences in instruction execution*. These differences can be used for CPU semantic attacks (see Section 2). Our contributions are:

1. We improve on the previously proposed Red Pill Testing [21, 20] by devising tests that carefully traverse operand space, and explore execution paths in instructions with the minimal set of test cases. We use 15 times fewer tests and discover 5 times more pills than Red Pill Testing. Our testing is also more efficient, 47.6% of our test cases yield a pill, compared to only 0.6% of Red Pill tests. In total, we discover between 7,487 and 9,255 pills, depending on the virtualization technology and the physical machine being tested.

2. We find two root causes of pills: (1) failure of virtual machines to strictly adhere to CPU design specification and (2) vagueness of the CPU design specification that leads to different implementations in physical machines. Only 2% of our pills stem from the second phenomenon.

3. We propose how to modify virtual machines to automatically hide presence of detected pills from malware, through introduction of additional interrupt vectors and by utilizing QEMU's interrupt handling mechanism for guest systems (Tiny Code Generation mode).

We emphasize that our testing methodology produces test cases selected at random from chosen input parameter ranges for each instruction – these ranges are chosen to exercise all execution paths in the given instruction's handling. If a test case's execution produces different outputs in a physical versus a virtual machine we say that this test case is a *pill*. While we only test one value from each parameter's range, if this test case is a pill, all values from the same parameter ranges would also lead to pills because they are all handled by the same path in that instruction's execution. Let us call a pill resulting from a test case a *test pill* and all related test cases that draw parameter values from the same input ranges as the test pill the *individual pills*. In this paper, all counts of pills we report are for test pills. Similar practice is adopted by related work [21, 20, 19]. The counts of individual pills are many times higher.

In Section 2 we give an overview of various anti-virtualization techniques. We survey related work in Section 3 and propose Cardinal Pill Testing in Section 4. We provide the details for the pills we find in Section 5 and analyze their root causes and completeness. In Section 6 we propose how to hide most of these pills from malware and we conclude in Section 7. All the scripts and test cases used in our study are publicly released at `http://steel.isi.edu/Projects/cardinal/`.

2 Anti-Virtualization Techniques

Anti-virtualization techniques can be classified into the following broad categories [8, 15]:

CPU Semantic Attacks. Malware targets certain CPU instructions that have different effects when executed under virtual and real hardware. For instance, the `cpuid` instruction in Intel IA-32 architecture returns the `tsc` bit with value `0` under the Ether [9] hypervisor, but outputs `1` in a physical machine [22]. As another example found in our experiment, when moving hex value `7fffffffh` to floating point register `mm1`, the resulting `st1` register is correctly populated as `SNaN` (signaling non-number) in a physical machine, but has a random number in a QEMU-virtualized machine. Malware executes these pills and checks their output to identify presence of a VM.

Timing Attacks. Malware measures the time needed to run an instruction sequence, assuming that an operation takes a different amount of time in a virtual machine compared to a physical machine [11]. Contemporary virtualization technologies (dynamic translation [5], bytecode interpretation [17], and hardware assistance [9]) all add significant delays to instruction execution that are measurable by malware [1].

String Attacks. VMs leave a variety of traces inside guest systems that can be used to detect their presence. For instance, QEMU assigns the "QEMU Virtual CPU" string to the emulated CPU and similar aliases to other virtualized devices such as hard drive and CD-ROM. A simple query to Windows registry would reveal the VM's presence immediately [8].

In this work we focus on handling the CPU semantic attacks as they are the most complex category to explore and enumerate. We note that string attacks can easily be handled through enumeration and hiding of VM traces, which can be done by comprehensive listing and comparison of files, processes and Windows registry with and without virtualization. Also, timing attacks can be handled through systematic lying about the VM clock, as proposed in [15]. While neither of these approaches

[1] This method can also be used to detect debuggers, because stepping code adds large delays.

is implemented today, both could be implemented as extensions of our work on lying to applications about CPU semantics (Section 6).

3 Related Work

Martignoni et al. present the initial Red Pill work in EmuFuzzer [21]. They propose Red Pill Testing—a method that performs a random exploration of a CPU instruction set and parameter spaces looking for pills. Testing is performed by iterating through the following steps: (1) initialize input parameters in the guest VM, (2) duplicate the content in user-mode registers and process memory in the host, (3) execute a test case, (4) compare resulting states of register contents, memory and exceptions raised—if there are any differences, the test case is a pill. In KEmuFuzzer [20], Martignoni et al. extend the state definition to include the kernel space memory, and test cases are embedded in the kernel to facilitate testing of privileged instructions. In their recent work [19], they use symbolic execution to translate code of a high-fidelity emulator (Bochs) and then generate test cases that can investigate all discovered code paths. Those test cases are used to test a lower-fidelity emulator.

While these works are seminal in pill detection they have several deficiencies that we seek to handle in this paper: (1) EmuFuzzer [21] tests boundary and random values for explicit input parameters, but does not cover implicit parameters. Their approach cannot guarantee that all types of pills will be detected. The symbolic execution approach [19] will discover differences between low-fidelity and high-fidelity emulators but not between an emulator and a physical machine. In addition, use of symbolic execution precludes test generation for floating-point instructions. We improve on these works by using instruction semantics to carefully craft test cases that explore all code paths. (2) Martignoni et al. use QEMU with Intel VT-x (in [21]) or Bochs emulator (in [19]) as an Oracle, while we use physical machines with no virtualization. This improves fidelity of testing and ensures detection of more pills.

Dinaburg et al. [9] aim to build a transparent malware analyzer, Ether, by implementing analysis functionalities out of the guest, using Intel VT-x extensions for hardware-assisted virtualization. nEther [22] work finds that Ether still has significant differences in instruction handling when compared to physical machines, and thus anti-virtualization attacks are still possible, i.e., Ether does not achieve complete transparency.

Kang et al. [15] propose a method to identify anti-emulation checks and modify virtual system states to "lie" to the malware, using semi-manual execution trace analysis. They record the malware trace in Ether, using it as an Oracle, and utilizing its debugging functions. They then automatically taint the variables in this trace, and manually identify those variables whose values are used in an anti-emulation check under QEMU. Their method requires manual intervention while we seek to overcome differences in execution environments automatically. Furthermore, since Ether is not identical to a physical machine, this approach will fail to detect some differences between a VM and a physical machine that we do detect.

Other works [25, 18, 2] focus on detecting anti-virtualization functions of malicious binaries based on profiling and comparing their behavior in virtual and physical machines. These works do not uncover the details of anti-virtualization methods that each individual binary employs, and they can only detect anti-virtualization checks deployed by their malware samples, while we detect many more differences that could be used in future anti-virtualization checks.

4 Cardinal Pill Testing

We now describe the architecture, test case generation and testing methodology for our Cardinal Pill Testing.

4.1 Architecture Overview

Our testing architecture is shown in Figure 1. It consists of three physical workstations: a master, a slave hosting a virtual machine (VM), and a slave running Windows 7 Pro x86 on a bare-metal as reference (Oracle). The slaves are connected to the master through two separate serial wires. The master is responsible for generating test cases (Section 4.3) and scheduling their execution in slaves. In both slaves, we configure an additional daemon in the testing system that helps the master set up a specific test case in each testing round.

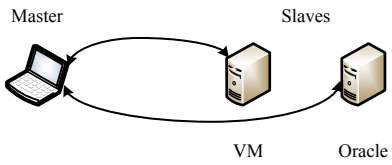


Figure 1: Architecture Overview

4.2 Logic Execution

The execution logic of our Cardinal Pill Testing is illustrated in Figure 2. The master maintains a debugger that issues commands to and transfers data back from the slaves. The Oracle and the VM have the same test case set and the daemon; we only show one pair of test case

and daemon in Figure 2 for clarity. We set the slaves in kernel debugging mode so that they can be completely frozen when necessary. At the beginning, the master reboots the slave (either VM or Oracle) for fresh system states. After the slave is online, the daemon signals its readiness to the master, which then evaluates test cases one by one in terms of rounds.

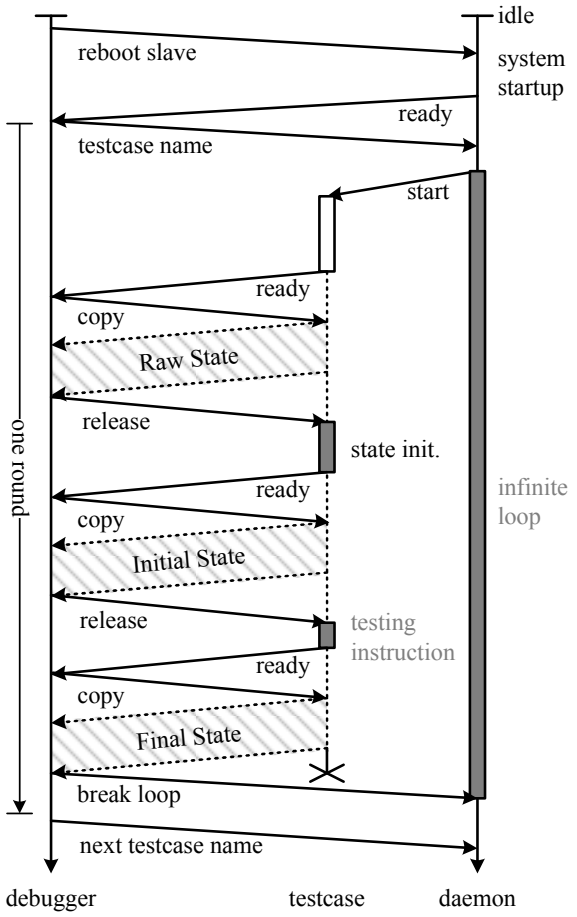


Figure 2: Logic Execution

We define the *state* of a physical or virtual machine as a set of all user and kernel registers, and the data stored in the part of code, data, and stack segments that our test case accesses for reading or writing.

In each round, the master interacts with the slave in three main phases. In the first phase, it issues a test case name to the daemon, which resides in a slave, and the daemon will ask the slave system to load this test case stored in its local disk. Then the system starts allocating memory, handles, and other resources needed by the test case program. After this *system loading* completes, the test case executes an interrupt instruction (`int 3`), which notifies the master and halts the slave. At this moment, the master saves the *raw state* of the slave locally. We use this raw state to identify *axiom pills* (see Section 4.3), instead of discarding it, as is done by EmuFuzzer [21] and KEmuFuzzer [20].

In the second phase, the master releases the slave which then executes the test case's initialization code and raises the second interrupt. Instead of using the same initial system state for all test cases, we carefully tailor register and memory bits for each test case, such that all possible exceptions and semantic branches can be evaluated (see Section 4.3). The master copies back the resulting *initial state* and releases the slave again.

In the third phase, the slave executes the actual instruction being tested and raises the last interrupt. The master will store this *final state* and use it to determine whether the tested instruction along with the initial state is a cardinal pill (see Section 5.1). It may happen that a test case drives the slave into an infinite loop or crashes itself or its OS. To detect this, we set up an execution time limit for each test case, so that the master can detect incapacitated slaves and restore them.

4.3 Test Case Generation

The quality of test cases is the key component of efficient pill discovery. The Red Pill work [21] generates test cases via two approaches: random generation and CPU-assisted generation. The former method randomizes data and code without conforming to any semantic rules, which may encode invalid instruction sequences. The latter combines each known opcode with some predefined operand values. Both approaches have the following deficiencies: (1) They only consider operands encoded in the instruction and fail to consider implicit arguments whose value may lead instruction execution to a different path in the code. For example, `rep stosb` takes no arguments but it depends on multiple register values. It stores contents of `al` register at the address specified by `es:(e)di`, and does this `ecx` times. Different values placed into those registers will result in different scenarios for `rep stosb` command use, such as writing into a valid versus invalid memory location, overwriting the instruction itself, using a zero, negative or very large positive value for the number of repetitions, etc. (2) They generate operands for instructions at random, which also does not explore all possible code paths. Our test case generation algorithm addresses both of these challenges.

4.3.1 Testing Goals

We aim to generate a minimal set of test cases for each instruction that explore all possible code paths in this instruction's handling. We start from the definitions of instruction handling recorded in a CPU manual. In this

work we focus on Intel's x86 CPU processor [13]. The manual details inputs and outcomes for each instruction, for normal execution and for exception handling. We call register modifications and exceptions whose semantics are fully defined in the manual *defined behaviors*. An instruction may also affect registers and raise exceptions that are specified in the manual as affected but the manner of their modification by the instruction is not specified. We call these modifications *undefined behaviors*.

For example, the `aaa` instruction adjusts the sum of two unpacked binary coded decimal (BCD) values to create an unpacked BCD result. The `al` register is the implied source and destination operand for this instruction. It also reads the `AF` flag in the `EFLAGS` register and writes to the `ah` register. Its normal execution will set `AF` and `CF` flags to `1` if the adjustment results in a decimal carry; otherwise it will set them to `0`. This is the defined behavior for the `aaa` instruction. In our testing we find that physical machines also set or reset the `SF`, `ZF`, and `PF` flags. While these flags are listed as affected by the instruction in the manual, there are no details of how they are calculated or what semantics they carry for the `aaa` instruction. This is the undefined behavior for the `aaa` instruction. In our work we explore both defined and undefined behaviors for each instruction, because both of these can be the source of pills.

Based on these observations, we set up the following goals of our test case generation algorithm:

- For defined behaviors for a given instruction, all branches should be evaluated. All flag bit states that are read implicitly or updated using results must be considered.
- All potential exceptions must be raised, such as memory access and invalid input arguments.
- Undefined behaviors should be investigated to reveal undocumented implementation specifics.

In the following sections, we first illustrate our test case template and then discuss how we group instructions and populate the template.

4.3.2 Test Case Template

We program a template to automatically generate test cases for most instructions, as shown in Figure 3. This program notifies the master and then halts the slave as soon as it enters the main function (line 2), so the master can save the states. The same interaction happens at lines 27, 29, and 38, after the test case completes a certain step. Then the program installs a structured exception handler for the Windows system (line 4 – 7). If an exception occurs, the program will ignore Windows' built-in exception handling routine and jump to line 31 directly, so we can save the system state before exception handling.

```
 1 main proc
 2   int 3               ; Raw State
 3
 4   push offset handler ; install SEH
 5   assume fs:nothing
 6   push fs:[0]
 7   mov  fs:[0], esp
 8
 9   ;; populate reg and memory
10   mov eax, 0000001bh
11   mov ebx, 00001000h
12   ...
13   ;; double precision floating-point
14   mov eax, 00403080h
15   mov dword ptr [eax], 0h
16   mov dword ptr [eax+4], 7ff00000h ; +Infi
17   ...
18   ;; single precision floating-point
19   mov eax, 0040318ch
20   mov dword ptr [eax], 0ff801234h  ; SNaN
21   ...
22   ;; double-extended precision FP
23   ...
24   ;; unsupported double-extended precision
25   ...
26   [state_init]      ; specific init
27   int 3             ; Initial State
28   [testing_insn]    ; instruction in test
29   int 3             ; Final State
30   call ExitProcess
31 handler:
32   ;; push exception information onto stack
33   mov edx, [esp + 4]        ; excep_record
34   mov ebx, [esp + 0ch]      ; context
35   push dword ptr [edx]      ; excep_code
36   ...
37   push dword ptr [edx + 0c0h] ; eflags
38   int 3           ; Final State (exception)
39   mov eax, 1h
40   call ExitProcess
41 main endp
42 end main
```

Figure 3: Test Case Template (in MASM assembly)

From line 9 to 25, we perform *general-purpose initialization*. Registers and memory are populated using pre-defined values, including all floating point and integer formats. This step occurs in all test cases and the carefully chosen, frequently used values, are stored in the registers to minimize the need for specific initialization. After this, the *specific initialization* (line 26) makes tailored modifications to the numbers, if needed for a given test case. For example, the `eax` is set to `1bh` at line 10 for all test cases. One particular test case may need `0ffh` value in this register and will update it at line 26. The actual instruction is being tested at line 29, where all defined and undefined behaviors use will be evaluated in various test cases. When compiling test cases, we disable

linker optimization and use a fixed base address, which does not affect testing but eases the interaction between the master and slaves.

4.3.3 Instruction Grouping

To test each instruction's behaviors in different execution stages, we need to vary the content in all registers and memory that the instruction reads. As discussed earlier and demonstrated in our evaluation in Section 5, random test generation cannot guarantee coverage of all code paths and execution branches. In our method, we manually analyze instruction execution flows defined in Intel manuals [13] and classify all possible input parameter values into ranges that lead to distinct execution flows.

IA-32 CPU architecture contains 906 instruction codes, and a human must reason about each to identify its inputs and outputs and how to populate them to test all execution behaviors. To reduce the scale of this human-centric operation, we first group the instructions into five categories: arithmetic, data movement, logic, flow control and miscellaneous. Arithmetic and logic category are further subdivided into general-purpose and FPU categories based on the type of their operands, We then define parameter ranges to test per category, and adjust them to fit finer instruction semantics as described below. This grouping greatly reduces human time investment and reduces chance of human errors. It took one person on our team a month and a half to devise all test cases. Table 1 shows the number of different mnemonics, examples, and parameter ranges we evaluate for each category.

Arithmetic Group. Instructions in this group first fetch arguments and then perform arithmetic operations. The arguments include actual data bits they operate on and certain flag bits that decide execution branches. We classify instructions in this group into two subgroups, depending on whether they work only on integer registers (general-purpose group), or also on floating point registers (FPU group). The instructions in the FPU group include instructions with x87 FPU, MMX, SSE, and other extensions.

Based on the argument types and sizes, branch conditions, and the number of arguments, we divide both subgroups into finer partitions. For example, `aaa`, `aas`, `daa`, and `das` in the general-purpose subgroup all compare the `al` register (holding one packed BCD argument 8-bits long) with `0fh` and check the adjustment flag `AF` in the `EFLAGS` register. This decides the output of the instruction. To test instructions in this set we initialize the `al` register to minimal (`00h`), maximal (`0ffh`), boundary (`0fh`), and random values in different ranges (`[01h, 0eh]`, `[10h, 0feh]`). We also flip `AF` between clear and set for different `al` values.

If a mnemonic takes two parameters, we select at least three value pairs to ensure that a greater-than, equal-to, and less-than relationship between them is satisfied in our test set. For the FPU subgroup, the parameter ranges are separated based on the sign, biased exponent, and significand, which splits all possible values into 10 domains: ±infi, ±normal, ±denormal, 0, SNaN, QNaN, and QNaN floating-point indefinite. We sample values from all these ranges to test behaviors in the arithmetic FPU group. For example, `fadd`, `fsub`, `fmul`, and `fdiv` each use one operand that can be specified using four different addressing modes; one of them is `m64fp`, which stands for a double precision float stored in memory. These instructions add/sub/mul/div the `st(0)` register with the operand's value and store the result in `st(0)`. In addition, they also read control bits in the `mxcsr` register and `fdiv` checks the divide-by-zero exception. In our test cases we generate values for the two floating point operands from the 10 identified ranges and permute the relevant bits in the `mxcsr` register. Because instructions in this subgroup can also access memory to read operands, we devise additional test cases to evaluate the memory management unit. We place the `m64fp` argument in and out of the valid address space of a data segment, into a segment with and without required privileges, and into a segment that is paged in and paged out of memory. By combining these test cases together, all potential memory access exceptions can be raised along with all potential arithmetic exceptions.

Data Movement. Data movement instructions copy data between registers, main memory, and peripheral devices and usually do not modify flag bits. There are several execution branches that we explore in tests. The source and the destination operands may be located outside segment limits. If the effective address is valid but paged out, a page-fault exception will be thrown. If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3, the system will raise an alignment exception. Some instructions also check direction and conditional flags, and a few others validate the format of floating point values. All these input parameters and the states that influence an instruction's execution outcome must be tested.

For example, we group 30 conditional movement instructions `cmov`*`cc`* `r32, r/m32` of distinct *`cc`* together because they move 32 bit signed or unsigned integers from the second operand (32 bit register or memory) to the first operand (32 bit register). The *`cc`* conditions are determined by the `CF`, `ZF`, `SF`, `OF` and `PF` flags. To access arguments outside the segment limit, we compile our test cases with the fixed base (Section 4.3.2). The starting addresses for code, data, and stack segment are `401000h`, `403000h`, and `12e000h` respectively, and each has a size of 4KB. It is difficult to test page faults

Category	Instruction Count	Example Instructions	Parameter Coverage
arithmetic	48	aaa, add, imul, shl, sub	min, max, boundary values, randoms in different ranges
	336	addpd, vminss, fmul, fsqrt, roundpd	±infi, ±normal, ±denormal, ±0, SNaN, QNaN, QNaN floating-point indefinite, randoms
data mov	232	cmova, fild, in, pushad, vmaskmovps	valid/invalid address, condition flags, different input ranges
logic	64	and, bound, cmp, test, xor	min, max, boundary values, >, =, <, flag bits
	128	andpd, vcomiss, pmaxsb, por, xorps	±infi, ±normal, ±denormal, ±0, SNaN, QNaN, QNaN FP indefinite, >, =, <, flag bits
flow ctrl	64	call, enter, jbe, loopne, rep stos	valid/invalid destination, condition flags, privileges
misc	34	clflush, cpuid, mwait, pause, ud2	analyze manually and devise dedicated input

Table 1: Instruction Grouping

directly because the Windows system does not provide APIs for page swapout. To work around this, we run other memory-consuming programs between test cases that use memory operands to force the values to be paged out of memory. In our evaluation, we find that this strategy works well and we successfully raise page faults when we need to test them. To raise the alignment checking exception, we store instruction operands at unaligned memory addresses. We permute the condition bits in the same way as we do for testing of arithmetic instructions.

Logic Group. Logic instructions test relationship and properties of operands and set flag registers correspondingly. We divide these instructions into general-purpose and FPU depending on whether they use EFLAGS register only (general-purpose) or they use both EFLAGS and mxcsr registers (FPU). We further partition logic instructions based on the flag bits they read and argument types and sizes. When designing test cases, in addition to testing minimal, maximal, and boundary values for each parameter, for instructions that compare two parameters we also generate test cases where these parameters satisfy larger-than, equal, and less-than conditions.

For example, one of the subgroups has bt, btc, btr, and bts instructions because all of them select a bit from the first operand at the bit-position designated by the second operand, and store the value of the bit in the carry flag. The only difference is how they change the selected bit: btc complements; btr clears it to 0; and bts sets it to 1. The first argument in this subgroup of instructions may be a register or a memory address of size 16, 32, or 64, and the second must be a register or an immediate number of the same size. If the operand size is 16, for example, we generate four input combinations (choosing the first and the second argument from 0h, 0ffffh values), and we repeat this for CF = 0 and CF = 1. Furthermore, we produce three random number combinations that satisfy less-than, equal and greater-than relationships. While the operand relationship does not influence instruction execution in this case, it does for other subgroups, e.g. the one containing the cmp instruction.

In the FPU subgroup, we apply similar rules to generate floating point operands. We further generate test cases to populate the mxcsr register, which has control, mask, and status flags. The control bits specify how to control underflow conditions and how to round the results of SIMD floating-point instructions. The mask bits control the generation of exceptions such as the denormal operation and invalid operation. We use ldmxcsr to load different values into mxcsr and test instruction behaviors under these scenarios.

Flow Control. Similar to logic instructions, flow control instructions also test condition codes. Upon satisfying jump conditions, test cases start execution from another place. For short or near jumps, test cases do not need to switch the program context; but for far jumps, they must switch stacks, segments, and check privilege requirements.

The largest subgroup in this category is the conditional jump *jcc*, which accounts for 53% of flow control instructions. Instructions in this group check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the required condition is satisfied they perform a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. In our test cases we vary the

status flags and set the relative destination addresses to the minimal and maximal offset sizes of byte, word, or double word as designated by mnemonic formats. For example, `ja rel8` jumps to a short relative destination specified by `rel8` if `CF = 0` and `ZF = 0`. We permute `CF` and `ZF` values in our tests, and generate the destination address by choosing boundary and random values from the ranges `[0, 7fh]` and `[8fh, 0ffh]`.

For far jumps like `jmp ptr16:16`, the destination may be a conforming or non-conforming code segment or a call gate. There are several exceptions that can occur. If the code segment being accessed is not present, a `#NP` (not present) exception will be thrown. If the segment selector index is outside descriptor table limits, an exception `#GP` (general protection) will signal the invalid operand. We devise both valid and invalid destination addresses to raise all these exceptions in our test cases.

Miscellaneous. Instructions in this group provide unique functionalities and we manually devise test cases for each of them that evaluate all defined and undefined behaviors, and raise all exceptions.

5 Detected Pills

We detect pills using our implementation of the architecture shown in Figure 1. We use two physical machines in our tests as Oracles: (**O1**) an Intel Xeon E3-1245 V2 3.40GHz CPU, 2 GB memory, with Windows 7 Pro x86, and (**O2**) Xeon W3520 2.6GHz, 512MB memory, with Windows XP x86 SP3. The VM host has the same hardware and guest system as the first Oracle, but it has 16 GB memory, and runs Ubuntu 12.04 x64. We test QEMU (VT-x), QEMU (TCG), and Bochs, which are the most popular virtual machines deploying different virtualization technologies: hardware-assisted, dynamic translation, and interpretation respectively. We allocate to them the same size memory as in the Oracle. We test QEMU versions 0.14.0-rc2 (**Q1**, used by EmuFuzzer), 1.3.1 (**Q2**), 1.6.2 (**Q3**), and 1.7.0 (**Q4**), and Bochs version 2.6.2. The master has an Intel Core i7 CPU and installs WinDbg 6.12 to interact with the slaves. For test case compilation, we use Microsoft Assembler 10 and turn off all optimizations. Our test cases take around 10 seconds to run on a physical machine and 15–30 seconds to run on a virtual machine.

Counting the different addressing modes, there are 1,653 instructions defined in the IA-32 Intel manual [13]. Out of these, there are 906 unique mnemonics. We generate a total of 19,412 test cases for these instructions.

5.1 Evaluation Process

We classify system states into user registers, exception registers, kernel registers, and user memory. The user registers contain general registers such as `eax` and `esi`. The exception registers are `eip`, `esp`, and `ebp`. The differences in the exception registers imply differences in the exceptions being raised. The kernel registers are used by the system and include `gdtr`, `idtr`, and others. In our evaluation, we do not populate kernel registers in the initialization step because this may crash the system or lead it to an unstable status. Further, initialization of kernel registers would require a system reboot and would make testing prohibitively expensive in a virtual machine. But, kernel register contents are saved as part of our states and compared to detect differences between physical and virtual machines.

For each test case, we first examine whether the user registers, exception registers, and user memory are the same in the Oracle and the virtual machine in the initial state. If they are different, it means that the VM fails to virtualize the initialization instructions (line 26 in Figure 3) to match their implementation in the Oracle. We mark this test case as "fatal" and discard it. If the initial values in these locations agree with each other, we then compare the final states. A test case will be tagged as a pill in two scenarios: (2) when the user registers, exception registers, and memory in the final states are different and (2) when the values in a certain kernel register are the same in the initial states but different in the final states.

5.2 Results

Table 2 shows the results of comparing various virtual machines to Oracle1 (O1).

The second column shows the number of pills for different virtual machines. Both QEMU (TCG) and Bochs exhibit moderate transparency—almost half of the test cases report different states between O1 and VMs. For Q2 (VT-x) 38.5% of our test cases result in pills, but there were no fatal cases. The pills we find for Q2 (VT-x) occur because QEMU does not preserve the fidelity provided by hardware assistance. Therefore, we should be careful when using hardware-assisted VMs for fidelity purposes. Their transparency depends on how they utilize the hardware extension.

The third column counts test cases that crash the system. For QEMU (TCG), one test case crashes the Oracle 1 and another one crashes the virtual machine. Another five crash both of them. For QEMU (VT-x) and Bochs, two test cases crash the physical and the virtual machine.

The number of fatal test cases are shown in the last column. All of them are related to FPU movement instructions. In some test cases that use denormals, SNaN, or QNaN values, the virtual machines could not populate the operand register as required. We note that we find no fatal test cases for VT-x technology.

Table 3 shows the breakdown of pills per instruction

VMs	pills	crash	fatal
Q1 (TCG)	9,255/47.7%	7/<0.1%	1,378/7%
Q2 (TCG)	9,201/47.4%	7/<0.1%	1,376/7.1%
Q1 (VT-x)	7,523/38.7%	2/<0.1%	3/<0.1%
Q2 (VT-x)	7,478/38.5%	2/<0.1%	0/0%
Bochs	8,958/46.1%	2/<0.1%	950/4.9%

Table 2: Results Overview

Category		Q1 (TCG)	Q2 (TCG)	Q1 (VT-x)	Q2 (VT-x)	Bochs	Total tests
arith	gen	877	872	633	626	920	2,702
	FPU	4,525	4,486	3,619	3,603	4,245	6,743
data mov		1,788	1,780	1,539	1,524	1,804	4,394
logic	gen	371	365	345	346	363	2,185
	FPU	1,446	1,447	1,132	1,127	1,362	2,192
flow ctrl		164	166	172	169	171	1,017
misc		84	85	83	83	93	179
total		9,255	9,201	7,523	7,478	8,958	19,412

Table 3: Pills per Instruction Category

category from Figure 1. The FPU arithmetic, FPU logic and data movement categories contain the most pills—around 83%. Table 4 shows the breakdown of the pills with regard to the resource that is different between a physical and a virtual machine in the final state. Most pills occur due to differences in the kernel registers.

5.2.1 Comparison with EmuFuzzer Pills

EmuFuzzer [21] generates 3 million test cases and the authors select 10% randomly to test in different virtual machines. The authors publish 20,113 red pills for QEMU 0.14.0-rc2 which is about 7% of the tested cases. Because they do not publish the entire test case set, we cannot directly compare our test cases with theirs, but instead we only compare the pills found by them and by us.

A *unique pill* is a pill whose mnemonic and parameter values do not appear in any other pill. We use the same QEMU version as EmuFuzzer (Q1 (TCG)) and run all the 20,113 red pills they found. We successfully extract operand values for 20,102 pills. After removing duplicate pills, there are 1,850 unique red pills (9%) and 136 different instruction mnemonics found by EmuFuzzer. Our 9,255 pills for Q1 (TCG) are all unique and there are 630 different instruction mnemonics. Furthermore, out of our 19,412 test cases we find 9,255 pills, which is 47.6% yield, while EmuFuzzer's yield is 1,850/300,000 = 0.6%. While direct comparison between our pills and EmuFuzzer's is difficult because both approaches select values of operands to test at random from specific ranges, we compare the ranges of the pills. This comparison shows that we detect all types of pills found by EmuFuzzer.

We conclude that our approach is more comprehensive than EmuFuzzer's and far more efficient. We cover all instruction mnemonics in our tests and find pills for 494 more instructions than EmuFuzzer. Overall we find five times more pills running 300,000/19,412 = 15 times fewer tests than EmuFuzzer. This illustrates the significant advantage of careful generation of operand values in tests over random fuzzing.

We further wanted to compare our pills with pills found by [19]. The Hi-Fi tests for Lo-Fi emulators [19] generate 610,516 test cases, out of which 60,770 (9.95%) show different behaviors in QEMU, and 15,219 (2.49%) show different behaviors in Bochs. Since the tests used for [19] are not publicly released we could not compare against them.

5.2.2 Root Causes of Pills

The differences detected by a pill can be due to registers, memory or exceptions that an instruction was supposed to modify, according to the Intel manual [13]. We call these instruction targets *defined resources*. However there are a number of instructions defined in the Intel manual that may write to some registers (or to select flags) but the semantics of these writes are not defined by the manual. We say that these instructions affect *undefined resources*. For instance, the `aas` instruction should set the `AF` and `CF` flags to `1` if there is a decimal borrow; otherwise, they should be cleared to `0`. The `OF`, `SF`, `ZF`, and `PF` flags are listed as affected by the instruction but their values are undefined in the manual. Thus the `AF` and `CF` flags are defined resources for the instruction `aas` and `OF`, `SF`, `ZF`, and `PF` flags are undefined.

Table 5 shows the number of pills that result from dif-

Category	Q2 (TCG)	Q2 (VT-x)	Bochs
user reg	2,416	34	1,671
excp reg	1,578	21	1,566
kerl reg	8,398	7,457	8,572
mem cont	46	9	20

Table 4: Details of Pills with Regard to the Resource Being Different in the Final State—in Some Cases Multiple Resources Will Differ so the Same Pill May Appear in Different Rows

ferences in undefined and defined resources for each instruction category compared to Oracle 1.

We note that a small number of pills that relate to general-purpose arithmetic and logic instructions occur because of different handling of undefined resources by physical and virtual machines. These comprise roughly 2% of all the pills we found.

For pills originating from defined resources, we analyze their root causes and compare them against those found by the symbolic execution method [19]. We find all root causes listed in [19] that are related to general-purpose instructions and QEMU's memory management unit.

In this work we do not extensively analyze pills that originate from differences in kernel-space handling of instructions, and thus cannot compare their root causes with those specified in [19]. Due to the extensive time required for testing (reboot is required after each test case) we leave this for future work.

Because the symbolic execution engine in [19] does not support FPU instructions, we discover additional root causes that are not captured by their method. First, we find that QEMU does not correctly update 6 flags and 8 masks in the `mxcsr` register when no exception happens, including invalid operation flag, denormal flag, precision mask, overflow mask. It also fails to update 7 flags in `fpsw` status register such as stack fault, error summary status, and FPU busy. Second, QEMU fails to throw five types of exceptions when it should, which are: float_multiple_traps, float_multiple_faults, access_violation, invalid_lock_sequence, and privileged_instruction. Third, QEMU tags FPU registers differently from Oracles. For example, it sets `fptw` tag word to "zero" when it should be "empty", and sets it to "special" when "zero" is observed in Oracles. Finally, the floating-point instruction pointer (`fpip`, `fpipsel`) and the data pointer (`fpdp`, `fpdpsel`) are not set correctly in certain scenarios. The details of all these root causes are given on our Web page.

5.2.3 Identifying Persistent Pills

Differences found in our tests between an Oracle and a virtual machine may not be present if we use a different Oracle or a different virtual machine, i.e. a difference may stem more from an implementation bug specific to that CPU or VM version than from an implementation difference that persists across versions. Furthermore, outdated CPUs may not support all instruction set extensions that are available in recent ones. Finally, recent releases of VM software usually fix certain bugs and add new features, which may both create new differences and remove the old differences between this VM and physical machines. We hypothesize that *transient* pills are not useful to malware authors because they cannot predict under which hardware or under which virtual machine their program will run, and we assume that they would like to avoid false positives and false negatives.

To find pills that persist across hardware and VM changes, we perform our testing on multiple hardware and VM platforms. We select 13 general instructions that can be executed in all x86 platforms (`aaa`, `aad`, `aas`, `bsf`, `bsr`, `bt`, `btc`, `btr`, `bts`, `imul`, `mul`, `shld`, `shrd`) and generate 2,915 test cases for them to capture more pills that are caused by modification of undefined resources. We evaluate this set on the two physical machines (Oracle 1 and Oracle 2), three different QEMU versions (Q2, Q3, and Q4), and Bochs. We find 260 test cases that result in different values in `EFLAGS` register in Oracle 1 and Oracle 2 and will thus lead to transient pills. Bochs' behavior for these test cases is identical to the behavior of Oracle 2. Out of the remaining 2,655 test cases, we find 989 persistent pills that generate different results in the three QEMU virtual machines when compared to the physical machines. They are all related to undefined resources. Bochs performs surprisingly well and does not have a single pill for these particular test cases. Thus we could not find persistent pills that would differentiate between any physical and any virtual machine in our tests but we found pills that can differentiate between any of the QEMU VM versions and configurations that we tested and any of the physical machines we tested.

We further investigate the persistence of pills that are caused by modifications to undefined resources, across different physical platforms. We select five physical machines with different CPU models in DeterLab [1]. Out of 195+23 = 218 pills that were found for Oracle 1 and Q2 (TCG) we were able to map 212 pills to all five physical machines (others involved instructions that did not

Category		Q2 (TCG)	Q2 (VT-x)	Bochs
arith	gen	195/677	0/626	194/726
	FPU	0/4,486	0/3,603	0/4,245
data mov		0/1,780	0/1,524	0/1804
logic	gen	23/342	0/346	20/343
	FPU	0/1,447	0/1,127	0/1,362
flow ctrl		0/166	0/169	0/171
misc		0/85	0/83	0/93

Table 5: Pills using Undefined/Defined Resources

Instruction	OF	SF	ZF	AF	PF	CF
aaa	0	0	ZF (ax)		PF (al + 6) or PF (al)	0
	0	0	ZF (al)		PF (al)	0
aad	F			F		F
	0			0		0
aam	0			0		0
aas	0	0	ZF (ax)		PF (al + 6 or al)	0
	0	0	ZF (al)		PF (al)	0
and, or, xor, text				0		
bsf, bsr	I	I		I	I	I
	0	0		F	0	0
bt, bts, btr, btc	I	I		I	I	
daa, das	0					
div, idiv	I	I	I	I	I	I
mul, imul		I	I	I	I	
		F	F	0	F	
		F	0	0	F	
rcl, rcr, rol, ror	I					
	F					
	OF(1-bit rotation)					
sal, sar, shl, shr shld, shrd	I			I		
	R			0		
	0			F		

Table 6: Undefined EFLAGS Behaviors

exist in some of our CPU architectures). Fifty of those were persistent pills—the undefined resources were set to the same values in physical machines. We conclude that modifications to undefined resources can lead to pills that are not only numerous but also persistent in both physical and virtual machines. This further illustrates the need to understand the semantics of these modifications as this would help enumerate the pills and devise hiding rules for them without exhaustive tests.

5.2.4 Completeness of Pills

Our test cases were designed to explore effects of input parameters on defined resources. We thus claim that our test cases cover all specified execution branches for user-space instructions and part for kernel instructions defined in Intel manuals. Our test pills should thus include all possible individual pills that can be detected for defined resources in user space. We cannot claim the same completeness for test pills that relate to defined or undefined resources in kernel space since we do not extensively test instructions that manipulate these resources, due to the reboot requirement.

We now further explore the pills stemming from modifications to undefined resources, to evaluate their impact on the completeness of our pill sets and to attempt to devise semantics of these modifications. The only undefined resources from the Intel manual are flags in the EFLAGS register.

We analyze the user-space instructions that affect one or more flags in the EFLAGS register in an undefined manner. We generate additional test cases for each instruction to explore the semantics of modifications to undefined resources in each CPU. Although the exact semantics differ across CPU models, we consider four semantics of flag modifications that are the superset of behaviors we observed across tested hardware and software machines: a flag might be (1) cleared, (2) remain intact,

(3) set according to the ALU output at the end of an instruction's execution, or (4) set according to an ALU output of an intermediate operation.

We run our test cases on a physical or virtual machine in the following manner. For each instruction, we set an undefined flag and execute an operation that yields a result inconsistent with the flag being set; for example, `ZF` is set while the result is `0`. If the flag remains set we conclude that the instruction does not modify it. Similarly, we can test if the flag is set according to the final result. If none of these tests yield a positive result, we go through the sub-operations in a given instruction's implementation, as defined in the CPU manual, and discover which one modifies the flag. For example: `aaa` adds `6` to `al` and `1` to `ah` if the last four bits are greater than `9` or if `AF` is set. The instruction affects `OF`, `SF`, `ZF` and `PF` in an undefined manner. We find that in some machines `ZF` and `PF` are set according to the final result, while in others `PF` is set according to an intermediate operation, which is `al = al + 6`.

Table 6 shows different semantics for each instruction, which are consistent across 5 different CPU models. Empty cells represent defined resources for a given instruction. Character "I" means the flag value is intact while "F" means that the flag is set according to the final result. Otherwise, the flag is set to the value in the cell.

To detect pills between a given virtual machine and one or many physical machines we repeat the same tests on the virtual machine, and look for differences in instruction execution semantics. If many physical machines are compared to a virtual machine we look for such differences where physical machines consistently handle a given instruction in a way that is different from how it is handled in a virtual machine. For example in Table 6, instruction `aad` either clears `OF`, `AF` and `CF` flags or sets them according to the final result. If a virtual machine were to leave these flags intact we could use this behavior as a pill.

Our test methodology will discover all test pills (and thus all possible individual pills) related to modifications of undefined resources by user-space instructions *for a given physical/virtual machine pair*. Since the semantics of undefined resource modifications vary greatly between physical CPU architectures, as well as between various virtual machines and their versions, all possible test pills cannot be discovered in a general case.

To summarize, our testing reveals pills that stem from instruction modifications to user-space or kernel-space registers. These modifications can further occur on defined or on undefined resources for a given instruction. We claim we detect all test pills (and thus all the individual pills) that relate to modifications of defined, user-space resources. We can claim that because we fully understand semantics of these modifications, and all physical machines we tested strictly adhere to this semantics as specified in the manual. We cannot claim completeness for pills that relate to modifications of undefined resources because physical machine behaviors differ widely for those. We further cannot claim completeness for pills that relate to modifications of kernel-space resources because we do not properly test initialization of these resources – such testing would require frequent reboots and would significantly prolong testing time.

5.2.5 Axiom Pills

In addition to comparing final states across different platforms we also compare raw states upon system loading. We define an *axiom* pill as a register or memory value whose raw state is consistently different between a physical machine and a given virtual machine. This pill can be used to accurately diagnose the presence of the given virtual machine. We select 15% of our test cases and evaluate them on Oracle 2, Q2, Q3 and Bochs. The axiom pills are shown in Table 7. For example, the value of `0ffffffffh` in the `edx` register can be used to diagnose the presence of Q2 (VT-x).

6 Improving Virtualization Transparency

EmuFuzzer [21] defines the virtualization transparency as how closely a virtual machine resembles the physical one. A perfect transparency means that programs in guests must not be able to tell if they are being executed in a virtual machine or not. The pills we find reflect the flaws of current virtual machine implementations, and specifically persistent pills reflect persistent flaws that can be used effectively by malware to detect virtualization. It would thus be desirable to develop techniques that hide the presence of reliable pills from malware. This could be achieved via multiple ways: (1) through patching of the current virtual machine implementations, (2) through overwriting of values in registers and memory with values consistent with physical machine deployment using kernel debuggers, (3) through modification of the guest OS so that malware reads of registers and memory after execution of pill instructions are intercepted and values consistent with physical machine deployment are returned (similar to kernel rootkit functionality), (4) through modifications of the host OS.

Out of all these approaches, patching VMs or guest OS are both time-consuming, may introduce other pills or bugs and do not apply to closed source implementations. Modifications to host OSes cannot hide all pills; for example in the TCG mode of QEMU, guest code translation happens in QEMU's user space, and the host cannot directly inspect guest instructions to detect pill execution. We thus choose to overwrite registers and memory

Reg	O1	Q1 (TCG)	Q2 (TCG)	Q1 (VT-x)	Q2 (VT-x)	Bochs
edx	vary	vary	vary	0ffffffffh	0ffffffffh	vary
dr6	0ffff0ff0h	0	0	0ffff0ff0h	0ffff0ff0h	0ffff0ff0h
dr7	400h	0	0	400h	400h	400h
cr0	8001003bh	8001003bh	8001003bh	8001003bh	8001003bh	0e001003bh
cr4	406f9h	6f8h	6f8h	6f8h	6f8h	6f9h
gdtr	vary	80b95000h	80b95000h	80b95000h	80b95000h	80b95000h
idtr	vary	80b95400h	80b95400h	80b95400h	80b95400h	80b95400h

Table 7: Axiom Pills

after pill instructions.

This overwriting can either happen in the virtual machine, through modification of VM code, or it could be performed by the same environment that is used for malware analysis, e.g. Anubis or Ether. We explore the first strategy here. We select QEMU TCG mode as our experiment platform since it has gained great popularity [6, 24, 4, 12]. We first explain how QEMU handles guest code translation and then describe how we integrate our pill hiding strategy into its translation code.

6.1 The Underhood of QEMU with TCG

Figure 4 describes two pivotal functionalities of QEMU: how TCG uses translation blocks to organize translated host code (x86 guest to x86_64 host in the example) and how QEMU executes translation blocks. A translation block is a consecutive memory of a few kilobytes located in a data segment, which consists of translated host code, prologue, and epilogue. It provides a full function layout as if generated from a compiler. As the name implies, the translated host code section stores host opcode generated by TCG, which acts as a function body. The prologue prepares the stack and registers for use within the function, while the epilogue restores the stack and registers to the state they were in before the function was called.

TCG translates guest instructions in two different ways. Simple guest instructions are mapped to host opcode directly; for example in Figure 4, the guest instruction `mov al, 8` is transformed to three host instructions. The actual translation operates at the opcode level without disassembling and compilation. For complex guest instructions, TCG uses helper functions to implement their semantics. For example, the guest `int` instruction will be replaced by a call to `helper_raise_int()`. Inside this function, QEMU checks the current CPU mode and then dispatches the interrupt. In dispatching, QEMU calculates the destination vector in the interrupt description table that should be selected. After the desired interrupt service routine is found, QEMU sets the guest code segment selector, offset, and instruction pointer, such that the guest will enter interrupt handling immediately after QEMU

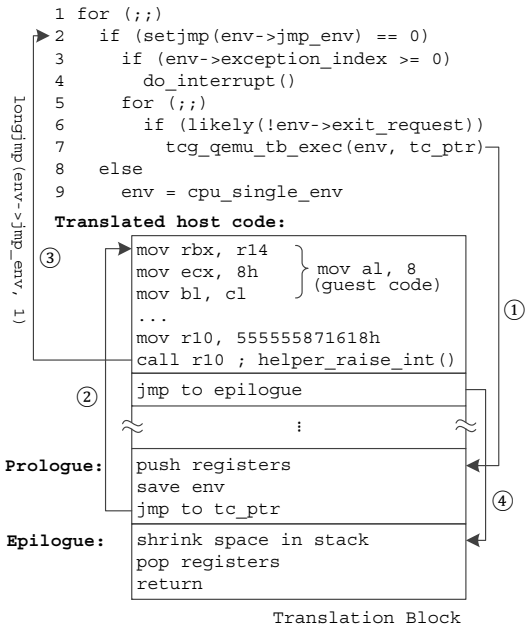


Figure 4: QEMU with TCG Translation and Execution.

yields control to the guest. Typically, QEMU stops translation if it encounters an `int` in the guest code, and `helper_raise_int()` will be the last instruction in a translation block in this case.

We summarize QEMU's main execution loop in lines 1–9 in Figure 4. It attempts to deliver all pending interrupts and exceptions and then finds the next translation block to execute. It takes advantage of the `setjmp()` and `longjmp()` facility provided by the C standard library to implement non-local jumps. At line 2, the QEMU context is saved to `jmp_env` by `setjmp()`. If this statement is actively called in place, line 3 will be examined and any pending interrupts and exceptions will be handled here. Otherwise, if the program flow returns here from a `longjmp()`, line 9 will be executed to reload CPU environment; then line 1 starts the next iteration. Lines 5–7 denote an infinite loop inside which QEMU repeatedly finds and executes translation blocks if no exception occurs. The function at line 7 is defined as a function pointer that is assigned to the

memory address of the prologue. At run-time, the prologue is cast as a function and executed (①) with parameters `env` and `tc_ptr`. The code bytes in the prologue save the current context and arguments to the stack. Then the program control will be transferred to the generated host code pointed to by `tc_ptr` (②). If the guest code contains an interrupt, the execution flow will follow the `helper_raise_int()` function generated by TCG (③); otherwise, this translation block will finish execution and step ④ is selected. In the first case, the helper function raises an interrupt with the vector number in the guest code, through setting of the corresponding data structures in QEMU. Then it calls `longjmp()` to jump to the latest context saved by `setjmp()`, so this function never returns. When executing line 2 following ③, the condition is not satisfied because `setjmp()` returns the argument `1` of `longjmp()`. Therefore, lines 3 – 4 will not be executed and the interrupt will not be repeatedly handled, which achieves the exact interrupt semantics. When the execution runs into the next round of the outside `for` loop, this pending interrupt will be handled in `do_interrupt()`.

6.2 Pill Hiding

Our proposed pill hiding strategy goes through three main stages: 1) detect pill instructions in the guest; 2) freeze the guest after the corresponding host code for the guest instruction has been executed; and 3) overwrite register and memory values using correct information learned from physical machines.

To detect pills, we need to compare the guest code with known pill instructions in run time. This can be achieved using either mnemonics or opcode. We choose the first approach since QEMU has a built-in disassembler.

To freeze the guest at the right point, we need to build a communication mechanism between QEMU and the guest. Debuggers achieve a similar functionality by replacing user-defined breakpoints with interrupt instructions. We cannot apply the same approach by inserting interrupts into translated code, since it will cause a trap between QEMU and the host instead. Actually, this is the reason why TCG needs to replace the guest interrupts with a call to the helper function as discussed in the previous subsection. To address this problem, we modify the QEMU's translation mechanism and utilize its interrupt handling mechanism as shown in Figure 5.

We monitor each guest instruction at line 1 by disassembling the current instruction in `pc_ptr`. If this instruction is not a pill, we directly translate it at line 11. If it is a pill, we check if the state before this instruction is saved. If not, this is the first time we encounter this instruction and we generate a `0x20` interrupt, otherwise we generate a `0x21` interrupt. Neither of these interrupt values are used by Windows. Generation of an interrupt calls `helper_raise_int()` in Figure 4 which brings the control to `do_interrupt()` as it does for other interrupt vectors. In this function we add new interrupt handlers for `0x20` and `0x21` interrupts. The handler for `0x20` saves the system state. The handler for `0x21` applies the hiding rules by overwriting the registers and memory with the values that a physical machine would set. The hiding rules can be devised by grouping pill instructions based on the resource that is the symptom of the pill (it is different in the physical and the virtual machine) and input parameter ranges. For example, we find 61 FPU instructions that always raise exceptions different from Oracles if their operands are in specific value ranges. When we detect these instructions and their operands fall in these specific ranges, we can raise the exceptions that occur in the Oracles. This would handle around 1,500 pills. Thus we can hide the presence of the pills without reimplementing instruction semantics. We emphasize here that only pills whose symptoms are not kernel registers can be hidden by our approach.

```
 1 curr_insn = disas(pc_ptr)
 2 if (curr_insn is pill)
 3     if (saved == false)
 4         gen_int(0x20)  // save states
 5         saved = true
 6     else
 7         pc_ptr = trans(pc_ptr)
 8         gen_int(0x21)  // apply hiding rules
 9         saved = false
10 else
11     pc_ptr = trans(pc_ptr)
```

Figure 5: Hooking on QEMU Translation

7 Conclusion

Virtualization is crucial for malware analysis, both for functionality and for safety. Contemporary malware tests if it is being run in VMs and applies evasive behaviors that hinder its analysis. Existing works on detection and hiding of differences between virtual and physical machines apply ad-hoc or semi-manual testing to identify these differences and hide them from malware.

In this paper we propose Cardinal Pill Testing that requires moderate manual action to identify ranges for input parameters for each instruction in a CPU manual, but then automatically that devises tests to enumerate the differences between a physical and a virtual machine. This testing is much more efficient and comprehensive than state-of-the-art Red Pill Testing. It finds five times more pills running fifteen times fewer tests. We further claim that for user-space instructions that affect defined resources, Cardinal Pill testing identifies all test pills that

could be used to generate all possible individual pills. Other categories contain instructions whose behavior is not fully specified by the Intel manual, which has led to different implementations of these instructions in physical and virtual machines. Such instructions need understanding of the implementation semantics to enumerate all the pills and devise the hiding rules. Our future work will focus on this direction. Yet other pills we have discovered stem from instructions that modify kernel-level resources. We do not properly test the initialization of these instructions because that would require reboot of machines and would be too time-consuming. Thus, we cannot claim completeness for pills that relate to kernel-level resources. We plan to test these extensively in our future work.

Acknowledgments

This material is based upon work supported by the Department of Homeland Security, and Space and Naval Warfare Systems Center, San Diego, under Contract No. N66001-10-C-2018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security for the Space and Naval Warfare Systems Center, San Diego.

References

[1] BAJCSY, R., BENZEL, T., BISHOP, ET AL. Cyber Defense Technology Networking and Evaluation. *Commun. ACM 47*, 3 (2004).

[2] BALZAROTTI, D., COVA, M., KARLBERGER, C., ET AL. Efficient Detection of Split Personalities in Malware. In *Network and Distributed System Security (NDSS)* (2010).

[3] BARFORD, P., AND BLODGETT, M. Toward Botnet Mesocosms. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets (HotBots)* (2007).

[4] BAYER, U., KRUEGEL, C., AND KIRDA, E. TTAnalyze: A Tool for Analyzing Malware. In *European Institute for Computer Antivirus Research (EICAR) Annual Conference* (2006).

[5] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC* (2005).

[6] BitBlaze: Binary Analysis for Computer Security. `http://bitblaze.cs.berkeley.edu/`.

[7] BRANCO, R. R., BARBOSA, G. N., AND NETO, P. D. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat* (2012).

[8] CHEN, X., ANDERSEN, J., MAO, Z., ET AL. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)* (2008).

[9] DINABURG, A., ROYAL, P., ET AL. Ether: Malware Analysis via Hardware virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)* (2008).

[10] FERRIE, P. Anti-Unpacker Tricks. `http://vpn23.homelinux.org/Anti-Unpackers.pdf`.

[11] FERRIE, P. Attacks on Virtual Machine Emulators. *Symantec Security Response* (2006).

[12] GOOGLE. Android Emulator. `http://developer.android.com/tools/devices/emulator.html`.

[13] INTEL. Intel 64 and IA-32 Architectures Software Developers Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[14] JOHN, J. P., MOSHCHUK, A., GRIBBLE, S. D., ET AL. Studying Spamming Botnets Using Botlab. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[15] KANG, M. G., YIN, H., HANNA, S., ET AL. Emulating Emulation-resistant Malware. In *Proceedings of the First ACM Workshop on Virtual Machine Security (VMSec)* (2009).

[16] KREIBICH, C., WEAVER, N., ET AL. GQ: Practical Containment for Measuring Modern Malware Systems. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC)* (2011).

[17] LAWTON, K. P. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 29es (1996).

[18] LINDORFER, M., KOLBITSCH, C., AND MILANI COMPARETTI, P. Detecting Environment-Sensitive Malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)* (2011).

[19] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 337–348.

[20] MARTIGNONI, L., PALEARI, R., FRESI ROGLIA, G., ET AL. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)* (2010).

[21] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., ET AL. Testing CPU Emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)* (2009).

[22] PÉK, G., BENCSÁTH, B., AND BUTTYÁN, L. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proceedings of the Fourth European Workshop on System Security (EuroSec)* (2011).

[23] SONG, C., ROYAL, P., AND LEE, W. Impeding Automated Malware Analysis with Environment-Sensitive Malware. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security (HotSec)* (2012).

[24] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.* (Hyderabad, India, Dec. 2008).

[25] SUN, M.-K., LIN, M.-J., CHANG, M., ET AL. Malware Virtualization-Resistant Behavior Detection. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)* (2011).

[26] YAN, L.-K., JAYACHANDRA, M., ZHANG, M., ET AL. V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)* (2012).

BareCloud: Bare-metal Analysis-based Evasive Malware Detection

Dhilung Kirat
University of California, Santa Barbara
dhilung@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Abstract

The volume and the sophistication of malware are continuously increasing and evolving. Automated dynamic malware analysis is a widely-adopted approach for detecting malicious software. However, many recent malware samples try to evade detection by identifying the presence of the analysis environment itself, and refraining from performing malicious actions. Because of the sophistication of the techniques used by the malware authors, so far the analysis and detection of evasive malware has been largely a manual process. One approach to automatic detection of these evasive malware samples is to execute the same sample in multiple analysis environments, and then compare its behaviors, in the assumption that a deviation in the behavior is evidence of an attempt to evade one or more analysis systems. For this reason, it is important to provide a reference system (often called bare-metal) in which the malware is analyzed without the use of any detectable component.

In this paper, we present BareCloud, an automated evasive malware detection system based on bare-metal dynamic malware analysis. Our bare-metal analysis system does not introduce any in-guest monitoring component into the malware execution platform. This makes our approach more transparent and robust against sophisticated evasion techniques. We compare the malware behavior observed in the bare-metal system with other popular malware analysis systems. We introduce a novel approach of hierarchical similarity-based malware behavior comparison to analyze the behavior of a sample in the various analysis systems. Our experiments show that our approach produces better evasion detection results compared to previous methods. BareCloud was able to automatically detect 5,835 evasive malware out of 110,005 recent samples.

1 Introduction

The malware threat landscape is continuously evolving. Early detection of these threats is a top priority for enterprises, governments, and end users. The widely-deployed signature-based and static-analysis-based detection approaches can be easily evaded by techniques commonly seen in the wild, such as obfuscation, polymorphism, and encryption. Therefore, dynamic malware analysis tools have recently become more popular to automate the analysis and detection of these threats [1, 14, 35]. These systems execute the suspicious sample in a controlled environment and observe its behavior to detect malicious intent. While this dynamic analysis approach is more effective against common static analysis evasion techniques, it faces a different set of challenges. More specifically, a malware sample, when executed, can detect the analysis environment and refuse to perform any malicious activity, for example by simply terminating or stalling the execution.

Malware authors have developed several ways to detect the presence of malware analysis systems. The most common approach is based on the fingerprinting of the runtime environment of the analysis system. This includes checking for specific artifacts, such as some specific registry keys, background processes, function hooks, or IP addresses that are specific to a known analysis tool. These artifacts must be known to the malware authors in advance to develop the corresponding fingerprinting techniques. Another approach leverages the fact that most of the analysis systems use emulated or virtualized environments as their malware execution platform. Such execution platforms can be detected by checking the platform-specific characteristics that are different with respect to a baseline environment (i.e., an unmodified operating system installed on real hardware, often referred to as a "bare-metal" installation). Such characteristics can be the timing properties of the execution, or a small variation in the CPU execution seman-

tics [31, 32].

Public-facing malware analysis systems are particularly vulnerable to the first approach to fingerprinting. This is because an attacker can submit malware samples specifically designed to extract the malware analysis environment artifacts to be then used in fingerprinting the analysis system. Private malware analysis systems are less prone to this type of fingerprinting. However, because of the internal sharing of malware samples among these private and public analysis systems, private systems may also be vulnerable to such fingerprinting [37].

One way to prevent the fingerprinting of the analysis environment is to construct a malware analysis system indistinguishable from a real host. Such systems are also known as *transparent* analysis systems. One of the first transparent analysis systems, called Cobra [34], tries to achieve this by developing a stealthy analysis environment using binary translation. However, this approach can only prevent known fingerprinting techniques. Ether [14] is a more robust transparent analysis system that leverages hardware virtualization to maintain the CPU execution semantics of a hardware CPU. However, the system introduces significant performance overhead when performing fine-grained monitoring, which is required to produce a comprehensive malware behavioral profile. With such performance overhead, it is fundamentally infeasible to make it transparent, especially if the malware execution has access to an external timing source [20].

Instead of preventing the fingerprinting of the analysis system, some of the recent works have focused on detecting a deviation of the malware behavior in different analysis environments [9, 13, 23, 24, 28]. The approach is to execute a malware sample in different analysis environments and compare their behavioral profiles to find a deviation. A behavioral profile is a higher-level abstraction of the activities performed by a malware sample when executed. The assumption is that the presence of such deviations is evidence of an attempt to fingerprint and evade one or more analysis systems. This is a generic and robust approach because it can detect evasion regardless of the knowledge of the techniques used by the malware sample in order to fingerprint and evade the analysis system. This approach assumes that the malware shows its malicious behavior in one of the analysis systems, also known as the *reference* system. However, all previous approaches have used emulated or virtualized environments for observing the deviation in the malware behavior, and such environments are known to be detectable. If all of the analysis systems are evaded by a malware sample, no significant deviation may be present in the execution traces. Moreover, some of the analysis systems use in-guest modules for behavior extraction, which further compromises the transparency of the analysis system.

A malware analysis system that is indistinguishable from a real host is a system that uses an unmodified operating system installation that runs on actual hardware (i.e., a bare-metal system). However, this approach faces several fundamental challenges. One of the important challenges is to efficiently restore the analysis system after every analysis run. Recently, a bare-metal-based malware analysis system, called BareBox [25], proposed an efficient system-restore technique. In this technique, the physical memory of the host is partitioned and only one partition is used for the analysis environment, while another partition is used for a snapshot of the system to be restored. Whenever needed, an external operating system located outside the physical memory of the analysis environment performs the restoration of the physical-memory snapshot, without the need for a reboot. However, a sophisticated malware can forcefully probe the physical memory and detect the presence of the BareBox system. Another bare-metal based malware analysis framework is Nvmtrace [5]. This system leverages IPMI (Intelligent Platform Management Interface) technology to automate the power cycle of the bare-metal analysis system. However, a complete reboot of the system is required after every analysis run. Another challenge to the bare-metal based malware analysis system is the extraction of the behavioral profile. To this end, no process-level behavior, such as process creation, termination, and hooking activities, can be extracted from a bare-metal analysis system without introducing some form of an in-guest analysis component. However, the presence of such components inside the system violates the transparency requirement and makes the system detectable. Because of this limitation, the observable malware behavior on a pure bare-metal system is limited to the disk-level and network-level activities. When only the disk-level and network-level behaviors are available, it may not be possible to perform an in-depth behavioral analysis, but these types of activity can be effectively used for detecting evasive behavior.

In this paper, we present BareCloud, a system for automatically detecting evasive malware. BareCloud detects evasive malware by executing them on a bare-metal system and comparing their behavior when executed on other emulation and virtualization-based analysis systems. Our bare-metal system has no in-guest monitoring component. This approach provides a robust transparent environment for our *reference system* where both user-mode and kernel-mode malware can be analyzed. BareCloud transparently extracts the behavioral profile of the malware from its disk-level and network-level activity. The disk-level activity is extracted by comparing the system's state after each malware execution with the initial *clean* state. Using the understanding of the operating system of the analysis host, BareCloud also extracts

operating-system-level changes, such as changes to specific registry keys and system files. Network-level activities are captured on the wire as a stream of network packets. This approach extracts malware behavior only from the persistent changes to the system. In principle, a malware sample could perform its activities without causing any persistent change, or could revert any changes after the activities are carried out. However, to perform any substantially malicious activity, a malware has to depend on some persistent change to the system, or it has to interact with external services, such as a C&C server. Both types of activities are transparently observable in our system.

When comparing the behavior of a malware sample on multiple platforms, previous works have considered the behavioral profiles purely as sets or bags of elements drawn from a flat domain, and computed their similarity using traditional set-intersection-based methods [8, 13, 28]. Set-intersection-based measures may not accurately capture similarity when data is sparse or when there are known relationships between elements within the sets [19]. For example, if two behavioral profiles under comparison contain a large number of similar file activities, but only one profile exhibits some network activities, set-intersection-based similarity measures, such as Jaccard similarity, produce a high similarity score, and fail to properly capture the lack of similarity among network activities. One may compute the similarity of the file activities and the network activities separately. However, similar problems exist; for example, two profiles may contain large number of similar DNS activities, but only one profile contains an HTTP request. It is important to identify such small-yet-important differences while comparing behavioral profiles for detecting evasions.

When manually comparing behavioral profiles, we start from generic questions such as "Do both profiles contain network and file activities?" If they do, we move on to other questions such as "Do these activities correspond to the same network or file objects?" This way of reasoning indicates that the behavioral profiles have an inherent similarity hierarchy based on the level of abstraction of the activities. Therefore, our similarity measure is based on the notion of the similarity hierarchy. Such hierarchy-base similarity can compute similarity at different levels of abstraction and identify activities that share similar characteristics even if they are not exactly the same. We show that this approach performs better than the set-intersection-based measure while comparing behavioral profiles for detecting evasive malware.

We compare the malware behavioral profile extracted from the bare-metal system with three major malware analysis platforms that are based on emulation and different types of virtualization, and we detect evasive behavior by detecting the deviation in the behavioral profile. Note that, beside evasion, there can be other factors that may cause a deviation in the behavioral profile. Section 4 describes how we mitigate those factors.

Our work makes the following contributions:

- We present BareCloud, a system for automatically detecting evasive malware. Our system performs malware analysis on a transparent bare-metal system with no in-guest monitoring component and on emulation-based and virtualization-based analysis systems.
- We introduce a novel evasion detection approach that leverages hierarchical similarity-based behavioral profile comparison. We show that this approach produces better results compared to the previous set-intersection-based approaches.
- We evaluate our system on a large dataset of recent real-world malware samples. BareCloud was able to detect 5,835 evasive malware instances out of 110,005 samples.

2 System Overview

The goal of our system is to automatically detect evasive malware by performing automated analysis of a large number of samples on a bare-metal reference system and other dynamic analysis systems. The goal is to identify deviations in the dynamic behavior of a sample when executed on different analysis environments. BareCloud achieve this by a multi-step process as depicted in Figure 1. The large volume of input samples is first pre-screened using the Anubis malware analysis framework [1]. The purpose of the pre-screening process is to select more interesting samples that are likely to have environment-sensitive behavior. These pre-screened samples are then executed on the cluster of bare-metal analysis hosts and on three other malware analysis systems, namely, Ether [14], Anubis [1], and Cuckoo Sandbox [2]. Each analysis system consists of multiple analysis hosts. The execution of the same sample in different systems is synchronized by the Scheduler component. Analysis hosts (workers) can independently join, perform analysis, and leave the BareCloud system. BareCloud extracts behavioral profiles from each of these analysis run, and, in the next step, it processes these profiles to detect evasive behavior.

3 Monitoring Environments

In this section, we describe the four malware analysis environments we use for monitoring the behavior of malware samples.

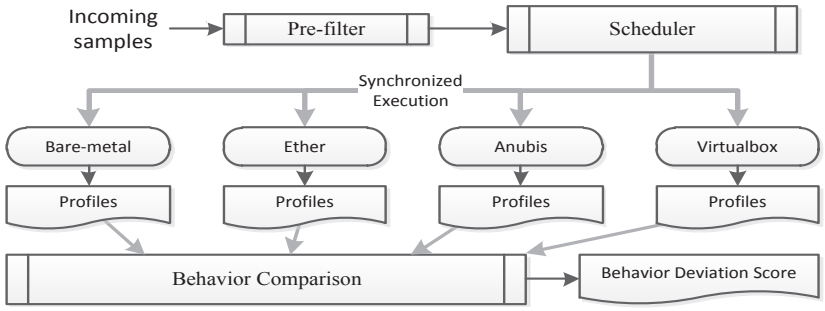


Figure 1: Overview of the system

3.1 Virtualization

We use Cuckoo Sandbox, which is based on VirtualBox [7], to provide a virtualization-based malware analysis platform. This is also known as a Type 2 hypervisor, that is, the virtualization host is a software application that runs on top of an operating system. Cuckoo Sandbox [2] supports the automation of the analysis process, and also includes function-hooking-based in-guest monitoring components. These components monitor execution-related events inside the analysis host. Several analysis and reporting plugins are available for specific needs. In this work, we use the analysis reporting plugin that includes Windows API call traces and network traffic. We use this trace to build the behavioral profile of a malware sample in the virtualized environment.

3.2 Emulation

We use the Anubis platform [1] to analyze malware in an emulated environment. Anubis is a whole-system emulation-based malware analysis platform. The emulator is based on Qemu [12]. No monitoring component is present inside the analysis environment other than some commonly used GUI automation tools. The emulator performs execution monitoring by observing the execution of pre-computed memory addresses. These memory addresses corresponds to important system API functions. Anubis is able to extract additional information about the API execution by inserting its own instructions to the emulator's instruction execution chain. Anubis implements a host of techniques, such as Product ID randomization, to prevent straightforward detection of the analysis system.

3.3 Hypervisor

We use Ether [14] to analyze malware in a hypervisor-based analysis environment. Ether is a Xen-based transparent malware analysis framework that utilizes Intel's VT hardware virtualization extensions [3]. The use of the hypervisor makes it possible to execute most of the malware instructions as native CPU instructions on the real hardware without any modifications. Thus, it does not suffer from inaccurate or incomplete system emulation issues that might affect emulation-based analysis systems. Ether can monitor a wide range of dynamic malware behaviors, such as system calls, memory writes, and fine-grained instructions execution. However, monitoring of memory writes and instruction-level trace introduces a substantial overhead and is only suitable for manual analysis. In this work, we only use Ether's coarse-grained system call trace collection capability. In addition, we also record all network communications during the malware execution. We combine the information from the system call trace and the network traffic to generate the behavioral profile.

3.4 Bare-metal

Our bare-metal malware analysis system is a cluster of hardware-based modular worker units. The workers' disks are based on a remote storage disk. This allows BareCloud to leverage copy-on-write techniques to perform disk restoration more efficiently when compared to a complete local disk overwrite. The bare-metal system also has a software-based remote control mechanism to automate the malware execution process on the workers. This mechanism is based on the IPMI remote administration features (e.g., IPMI allows to control the power cycle of the analysis worker units). We use the iSCSI protocol (Internet Small Computer System Interface) to attach remote disks to worker units. We used Logical Volume Manager (LVM)-based copy-on-write snapshots to host the remote disk used by the analysis system running on the worker units. After the completion of each malware analysis, the corresponding volume snapshot is recreated from a clean volume.

One of the critical components of a malware analy-

sis system is the *malware initiator*, which is the component that starts the execution of the malware. Usually, this component is implemented as some form of in-guest agent that waits for a malware sample through a network service. If the analysis system reboots after each analysis run, another approach can be to install a start-up entry in the system configuration that executes an application from a specific path. A malware sample can then be updated at this specific path for each analysis run by directly modifying the disk-image when the analysis system is offline. However, precise control over the malware execution duration is difficult when using this approach, as the overall execution time includes the system reboot time, which can vary among multiple reboots..

For a bare-metal analysis system, making its *malware initiator* component transparent is very important. This is because the malware can simply check for the presence of this component to fingerprint the environment. To this end, our system uses the network-based approach. The *malware initiator* removes itself and all of its artifacts after initiating the malware. This network-based approach also makes the malware execution duration more accurate, as it does not account for the reboot time.

3.5 User Environment

Apart from stock operating system, the environment installed inside the malware analysis systems includes some data and components that are usually present on a real host, such as saved credentials for common social networks, browser history, user document files, and other customizations. With this setup, we can observe additional malware behavior that we could not have observed using a bare user environment.

4 Behavior Comparison

In this section, we discuss malware behavioral deviation, behavioral profile extraction, and formalize behavioral profile comparison.

4.1 Behavior deviation

There are many factors that may cause a malware sample to show deviations in the dynamic behavior associated with different analysis environments. Hereinafter, we discuss each of these factors in detail.

- Evasive behavior of the malware sample:

  Deviation in the behavior may be the result of a successful fingerprinting of the analysis environment. This deviation is observable due to the change in the activities performed by the malware after the detection. This is the type of deviation we are interested in.

- Intrinsic non-determinism:

  A malware may have intrinsic non-determinism embedded in the code. That is, malware behavior might depend on some random value that it reads at the time of execution. For example, a malware sample may create a file with a random name. Randomization in the behavior can also result from the use of certain system services and APIs. For example, a successful call to `URLDownloadToFile` creates a random temporary folder to download the web content.

- Internal environment:

  Difference in the software environment of the different analysis systems may trigger different dynamic behaviors of the malware sample. For example, some malware may depend on a .NET framework installed in the analysis system, or may depend on the availability of a specific version of a system DLL. If one of the malware analysis environments does not contain such software components, the resulting malware behavior may be different.

- External environment:

  Another critical factor that may cause a deviation in the malware behavior is the external environment with which a malware sample can interact. In the context of malware execution, this external environment largely comprises of different network services, such as DNS and C&C servers. The non-deterministic nature of such network services may introduce deviations in the dynamic behavior of a malware sample. One simple way to minimize this factor is to completely disable access to external network environments. However, the network activity of a malware sample is one of the most important aspects to characterize the behavior of the sample. Hence, a successful behavior comparison of a malware sample requires the inclusion of its network activities.

Since our goal is to identify behavior deviations caused by the evasive technique employed by the malware sample, we need to minimize the effect of the three other factors that may cause a behavior deviation.

One approach to identifying intrinsic non-determinism is to execute the same sample in the same environment multiple times. By comparing the execution traces from

these different execution runs, non-deterministic behavior can be identified. Previous work [28] used this approach to filter out randomized behavior. However, this approach is resource- and time-expensive. Moreover, not all malware exhibit such randomized behavior.

In this work, we propose a more efficient hierarchical similarity-based approach to behavior comparison, described in Section 4.4. This approach is able to minimize the effect of intrinsic randomization without requiring multiple execution runs of the same sample in the same analysis environment. In order to address deviation caused by different internal environments, we must provide identical software environments to all analysis systems. Therefore, we prepared identical base software environments for all of our analysis systems.

Precisely controlling the behavior deviation introduced by the external environment is difficult. This is because these factors are not under our direct control. However, failure to minimize the impact of these factors may result erroneous behavior deviations. This consideration is important because most malware communicates with the external environment to carry out its malicious activities. To minimize the effect of the external environment, we implemented the following strategies.

- Synchronized execution: We execute the same malware sample in all analysis environments at the same time. The scheduler component facilitates the synchronization among different analysis hosts. By doing this, we minimize the behavior deviation that may be introduced by the variation of the external factors over time. For example, a malware may try to connect to a *fast-flux* network. The availability and the returned response of the C&C server and the DNS server may vary over time. If the malware is executed in different environments at different times, such variations in external environment may result in a spurious behavior deviation. Synchronized execution mitigates such differences.

- Identical local network: Malware can interact with the local network by different network-related activities, such as probing available local network services and accessing file shares. We expose all analysis systems to identical simulated local network environments.

- Network service filters: One approach to minimize the non-determinism introduced by different network services is to actively intercept network communications and maintain identical responses to identical queries among all instances of a malware running in different analysis environments. This requires an application-level understanding of the network services. To this end, we intercept all DNS and SMTP communications and respond with consistent replies in all analysis system. For example, the system responds with identical IP information to identical DNS queries coming from different analysis environments. With this setup, we are also able to sinkhole non-existent domain and SMTP communications to the local simulated network. This helps us observe more network behavior of a malware sample, which otherwise may not be observable.

4.2 Behavioral profile

After the execution of a malware sample in different analysis environments, we need to extract its behavioral profile for comparison. Usually, the behavioral profile is extracted from some form of dynamic execution trace, such as a system-call trace. Bayer et al. have introduced a comprehensive method of extracting behavioral profile from an augmented system-call trace. The additional information provides taint tracking of input and output parameters of system calls that provides dependency information between different system calls [10]. This approach has been used to cluster a large number of malware, and to compare malware behaviors [10, 28]. Similar approaches can be used in three of our analysis environments, where system-call traces are available. However, this system-call based approach is not directly applicable to our bare-metal malware analysis system, as we do not have access to the system-call trace.

Transient and resultant behavioral profile

A transient behavioral profile is a profile that represents all of the operations performed by a malware sample during its execution. The system-call-based behavioral profile discussed previously is a type of transient behavioral profile. This represents a more comprehensive view of *how* a malware performs its malicious activities. The resultant behavioral profile consists of the cumulative changes made by the malware from the beginning to the end of its execution. This includes those operations that make persistent changes to the system. Multiple similar operations to the same object are combined and represented as one operation to reflect the resulting effect of the operations. This represents a more summarized view of *what* a malware does to the system. A malware can obfuscate its transient behavior to evade transient behavior-base similarity detection. However, similar malicious activities produce similar resulting behavioral profiles, even if the transient behavior is obfuscated or randomized. This makes the comparison of malware behavior based on the resultant behavioral profile more robust.

The transparency requirement of our bare-metal analysis system limits us to the extraction of only the resulting behavioral profile. That is, the transient behaviors of process activities and filesystem activities are not available. However, we can extract the resulting filesystem behavior by comparing the disk contents from before and after the malware execution. Extraction of network behavior is straightforward using an external traffic capture component.

With these constraints in hand, we model our behavioral profile based on the model introduced by Bayer et al. [10], such that only the objects and the operations are used. That is, we take into consideration the object upon which a malware performs an operation that causes a persistent change to the object. Formally, a behavioral profile Π is defined as a 3-tuple.

$\Pi = (O, R, P)$

Where, O is the set of all objects, R is the set of all operations that causes persistent changes, and $P \subseteq (O \times R)$ is a relation assigning one or more operations to each object. Unlike in the model proposed in [10], where the objects and the operations are conceptualized as OS Objects and OS operations, we generalize the objects and operations to any environment entity with which a malware can interact. More details on objects and operations are provided hereinafter.

Objects

An object represents an entity, such as a file or a network endpoint, upon which a malware can perform some operation.

It is a tuple of `type` and `name` formally defined as follows.

$O = (obj\_type, obj\_name)$

$obj\_type ::= file|registry|sysconf|mbr|network$

The *file* type represents filesystem-specific file objects of the disk, the *registry* type represents registry keys, the *sysconf* type represents OS-specific system configurations, such as the boot configuration, *mbr* represents OS-independent Master Boot Record, and the *network* type represents network entities, such as a DNS server.

Operations

An operation generalizes the actions performed by a malware sample upon the above-described objects. An operation is formally defined as:

$R ::= (op\_name, op\_attribute)$

That is, an operation has a name and a corresponding attribute to provide additional information. As mentioned previously, only those operations that cause a persistent change to the system are included. For example, in case of a *file* type object, only the creation, deletion, and modification operations are included in the profile.

4.3 Behavior extraction

Our bare-metal system can only access the raw disk contents. We extract the filesystem behavior by comparing the filesystem state before and after the execution of a malware sample. A detailed understanding of the filesystem internal structures is required to extract such information. We leverage the functionalities provided by the SleuthKit framework [6] for extracting the file meta-data information from the raw disk image. By doing this, we are able to extract all file names in the disk, including some recently deleted files, along with their corresponding meta-data, such as size and modification date. We first build two sets representing the file object meta-data: the *clean* set and the *dirty* set, corresponding to the disk content before and after a malware execution. Extracting the deletion and creation operations of a file object are simple set operations. That is, any file not present in the dirty set is considered as deleted, and any file only present in the dirty set is considered as created. If a file is present in both sets with different meta-data, it is considered as modified. However, if a malware writes to a disk-sector (other than MBR) that is invisible to the filesystem, or modifies an existing file without changing the size and file-date meta-data, the current approach will not detect such changes. The straightforward way of comparing all file contents between two disk states can be very inefficient. This limitation can be mitigated by first detecting such changes in the disk sectors from copy-on-write data or iSCSI communication, and mapping the dirty sectors to files. Similar approach has been previously proposed [29]. To this end, we leave this improvement as a future work.

Registry behavior is extracted using a similar approach. We extract the meta-data of all the registry keys from the raw registry hive (registry database file) using the *registryfs* filesystem extension of the SleuthKit framework. Again, we build two sets representing the registry meta-data corresponding to the registry hive content before and after the malware execution. We perform set operations similar to the case of the filesystem to extract malware operations on the registry objects.

To extract the behavior of type *sysconf*, we process the filesystem and registry behavior to identify critical modifications to the system configuration. Some examples of the system configuration locations are listed in Table 1.

For the three other analysis systems, we process system-call traces to extract behavior information.

Table 1: Examples of the configuration locations

obj_type	obj_name	System path
sysconf	startup	HKLM/Software/Microsoft/Windows/CurrentVersion/Run
sysconf	startup	HKCU/Software/Microsoft/Windows/CurrentVersion/Run
sysconf	startup	HKLM/System/CurrentControlSet/Services
sysconf	boot	%SYSTEMROOT%/BOOT.INI
sysconf	autoexec	%SYSTEMROOT%/AUTOEXEC.BAT
sysconf	sysini	%SYSTEMROOT%/WINDOWS/SYSTEM.INI
sysconf	winini	%SYSTEMROOT%/WINDOWS/WIN.INI

Behavior normalization

Behavioral profile extracted from the difference of the initial and final disk states contains both malware behavior as well as the background operating system behavior. We need to filter out the features of the behavioral profile that do not correspond to the malware execution. One way to filter such features is to use the file modification timestamp of the file objects. That is, by selecting only those files that are created and modified during the time when the malware is executed, one can filter out unrelated file modifications that occur before and after the malware execution. However, some unrelated filesystem changes caused by the *base* operating system might still be present in the filtered profile. Moreover, many malware samples actively modify the system time, or tamper with the file meta-data to revert the file's modification time. Although the simple file time-stamp-based filter is efficient, this approach will fail in such situations.

Another approach to filter the background behavior is to first learn the behavioral profile of the *base* operating system and then filter this behavior from the profile generated by a malware execution. By doing this, we can overcome many of the shortcomings of the timestamp-based approach. This approach may exclude some malware operations that match the operation performed by the *base* operations. However, it is difficult to perform malicious actions using only operations that are also performed by the base operating system. Also, such operations are less important in defining the malicious behavior of the malware. We use this approach to filter our profiles. To extract the background behavior of the analysis system, we wrote a "void" program that does nothing other than stall infinitely. For each analysis environment, we extract the behavioral profile of the "void" program from all of its analysis hosts and combine them to build a generalized background profile. We use this profile to filter the behavioral profile of a malware execution.

Some objects used to describe the profile may be referenced using multiple names. For example, `\\?\C:\Documents and Settings` and `C:/DOCUME~1/` correspond to the same file object. We convert such identifiable object names to the same format. Different usernames may also result in different physical names for semantically similar file locations. For example, the locations `C:/DOCUME~1/USERA` and `C:/DOCUME~1/USERB` are semantically similar locations, which is the user's home directory. Some system APIs that create temporary files also generate different file paths, which are semantically similar. Many such temporary path names have known root locations and can be identified by their naming structure. We replace such occurrences in the object names with corresponding generic tokens.

4.4 Behavior comparison

Previous works have compared the persistent change-based behavioral profile using set-intersection-based methods over the feature set [13, 28]. However, when comparing behavioral profiles that only considers persistent changes, one can expect a sparse feature set. Furthermore, features within the profile are highly related and can be categorized in groups and subgroups. However, when the features are sparse or when there are known relationships between features within the set, set-intersection-based measures may not accurately capture the similarity [19].

Unlike previous works, we use a hierarchical similarity measure to overcome this problem. The hierarchy is associated with the different abstraction levels present in the behavioral profile. This approach makes our similarity measure less sensitive to randomization introduced by non-determinism in malware code. This is because the randomization is usually introduced only in one level of the hierarchy while keeping other levels of the hierarchy identical. For example, a malware may randomize the filename (obj_name) it creates, but perform the same *create* operation (op_name) on a *file* object (obj_type) with the same operation *attribute* ($op\_attribute$).

4.5 Hierarchical similarity

The notion of the hierarchical similarity is often used in text similarity, in mining association rules, and in various computer vision tasks for finding similar shapes [16, 17, 21]. We use a similar notion of hierarchical similarity to compare behavioral profiles. The similarity hierarchy of the behavioral profile is represented in Figure 2. As one can see, knowledge of the semantics and of the relationship between the objects is encoded in the representation. The leaves of the tree are the actual feature elements of the behavioral profile. The first level of similarity hierarchy is obj_type. An obj_type may have one or multiple obj_name, and each such obj_name can be associated with one or more op_name corresponding to various operations. Each such operation has one leaf node corresponding to the associated attribute of the operation. The leaf nodes are the feature elements whose attributes are represented by its parent nodes. For example, in the Figure 2, the ele-

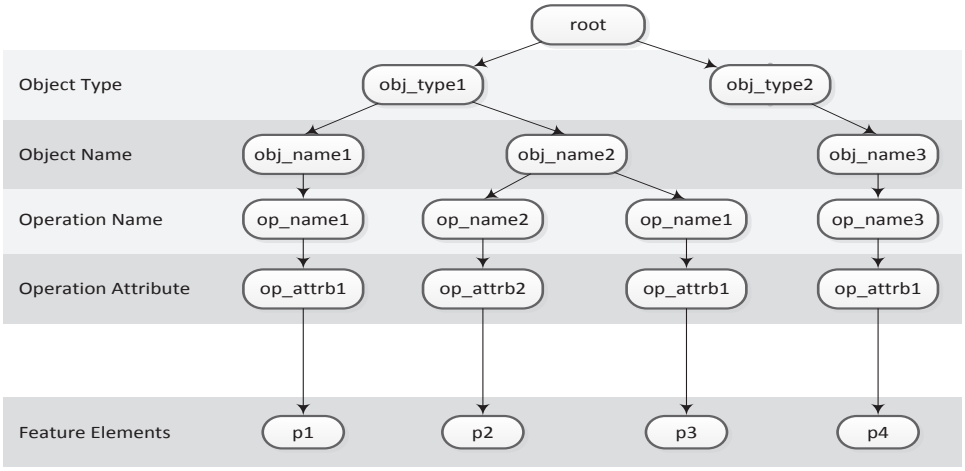


Figure 2: Behavior similarity hierarchy

ment p_1 is a feature element having the feature attributes $(obj\_type1, obj\_name1, op\_name1, op\_attrb1)$.

We compute the similarity in a two-step process. First we identify the matching nodes in the hierarchies of two behavioral profiles. We do this iteratively, starting from the first level (obj_type). For each of these matching nodes, we identify the matching nodes among their child nodes, i.e., the next level of hierarchy. We compute the similarity measure at each hierarchy level. Finally, we aggregate level similarity measures to compute the overall similarity.

The model

Let H be a rooted tree representing the similarity hierarchy, where all nodes have associated labels. For example, Figure 2 is an instance of H. Let L_H be all labels in H, and $L(H,d)$ be the set of labels of the nodes of H at depth d. Let δ be the height of the tree such that $L(H,\delta)$ is the set of all labels of the leaves of H. The set of labels $L(H,\delta)$ represents the feature elements p such that $p \in P$, where P represents the behavioral profile. In the example Figure 2, the leaf nodes p_1, p_2, p_3, and p_4 represents the feature elements of the behavioral profile. $L(H,\delta)$ and P are equivalent, one represented as leaves of the tree structure, another represented as a tuple of the feature attributes. With this, any P can be mapped into H. There is a hierarchy in $L(H,\delta)$, and hence in P, superimposed by H.

Let P_1 and P_2 be two behavioral profiles of a malware sample m from analysis systems a_1 and a_2. Let these behavioral profiles be mapped into hierarchies H_1 and H_2, instances of the hierarchical model H. Let $PL(H,l)$ be the label of the parent node of a node with label l, where $l \in L_H$, the set of all labels in H. Here, we want to find nodes with matching labels at each depth d whose parent nodes also have matching labels. We recursively define $match$ and $candidate$ for each level d as:

$$match_{H_1,H2}(d) = L(H_1,d) \cap L(H_2,d) \mid \forall l \in match_{H_1,H2}(d), PL(H_1,l) = PL(H_2,l) \text{ and } PL(H_1,l) \in match_{H_1,H2}(d-1) \quad (1)$$

$$candidate_{H_1,H2}(d) = L(H_1,d) \cup L(H_2,d) \mid \forall l \in candidate_{H_1,H2}(d), PL(H_1,l) \cup PL(H_2,l) \in match_{H_1,H2}(d-1) \quad (2)$$

where,

$$match_{H_1,H2}(0) = root. \quad (3)$$

We define $levelsim_{H_1,H_2}(d)$, the similarity of H_1 and H_2 at level d, as the Jaccard similarity coefficient. That is,

$$levelsim_{H_1,H_2}(d) = \frac{|match_{H_1,H2}(d)|}{|candidate_{H_1,H2}(d)|}. \quad (4)$$

We define the overall hierarchical similarity between behavioral profiles P_1 and P_2 as the arithmetic average of similarity at each level:

$$Sim(P_1,P_2) = \frac{1}{\delta}\sum_{d=1}^{\delta-1} levelsim_{H_1,H_2}(d). \quad (5)$$

This definition is consistent, since the right side of this equation always lies between 0 and 1. Hence, the *behavior distance* between P_1 and P_2 can simply be defined as:

$$Dist(P_1,P_2) = 1 - Sim(P_1,P_2). \quad (6)$$

This is possible because $Sim(P_1,P_2)$ is derived from the Jaccard similarity coefficients.

Finally, we define the behavior *deviation score* of a malware sample D among different analysis system $a_1 ... a_n$ with respect to the behavioral profile P_r extracted from the reference system a_r as the quadratic mean of the behavior distances as follows.

$$Deviation(D) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} Dist(P_r,P_i)^2}, \quad (7)$$

where n is the number of analysis systems, and P_i is the behavioral profile extracted from the analysis system a_i. This deviation score D is in interval [0,1], where the value of 0 means no deviation and the value of 1 means maximum deviation. We define a deviation threshold t. If the deviation score D exceeds this value, we consider the sample as evasive.

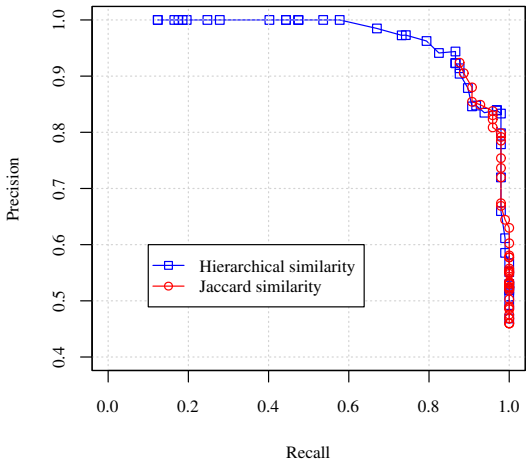


Figure 3: Precision-Recall analysis of the hierarchical similarity-based and the Jaccard similarity-based behavior comparison.

5 Evaluation

We evaluated our approach using Windows XP SP2 as the operating system for all analysis systems, as the weaker security measures of this operating system allow us to observe more malware behavior.

We conducted two experiments, which are detailed in the following sections.

5.1 Experiment I

In this experiment, we evaluated our hierarchical similarity-based behavior comparison approach.

Dataset

We first built the ground truth dataset of evasive malware and non-evasive malware samples. We received 234 recent and possibly evasive malware samples from a security company. We manually analyzed them and confirmed 111 samples from 29 families to be evasive (i.e., they fingerprint and evade at least one of the considered analysis environments). To build the dataset of non-evasive samples, we manually analyzed recent samples submitted to Anubis. By doing this, we selected 119 samples from 49 families that did not exhibit evasive behavior in any of the analysis environments.

We extracted the behavioral profiles of these samples from all analysis environments and computed the hierarchical similarity-based deviation score D with respect to the bare-metal analysis environment. We also computed the Jaccard distance-based deviation score as proposed in [28]. That is, the behavioral profile distance is computed as the Jaccard distance:

$$J(P_1,P_2) = 1 - \frac{P_1 \cap P_2}{P_1 \cup P_2}.$$

The Jaccard distance-based deviation score JD is computed as the maximum of the distances among different analysis systems as:

$$JD = \max_{1<i<n} \{ \max_{1<j<n, i\neq j} \{J(P_i,P_j)\}\}.$$

Since we do not extract multiple behavioral profiles of same malware sample from the same analysis environment, we did not compute the *intra-sandbox distance* as proposed in [28]. Instead, we wanted to compare the two different similarity approaches when the available behavioral profiles are limited to one profile per analysis environment.

We performed a precision-recall analysis by varying the threshold t for these deviation scores. If the deviation score is greater than t, the sample is considered evasive. The result is presented in Figure 3. We can clearly see that the hierarchical similarity-based deviation score gives better results. It is able to produce higher precision results while maintaining a high recall rate. While the highest precision the regular Jaccard distance-based similarity could produce is limited to 88.54%. This is because some non evasive samples have the same Jaccard distance-based deviation score as other evasive samples having highest deviation score. Hence, no matter how we vary the threshold t, precision does not increase.

Figure 4 shows the precision-recall characteristics of

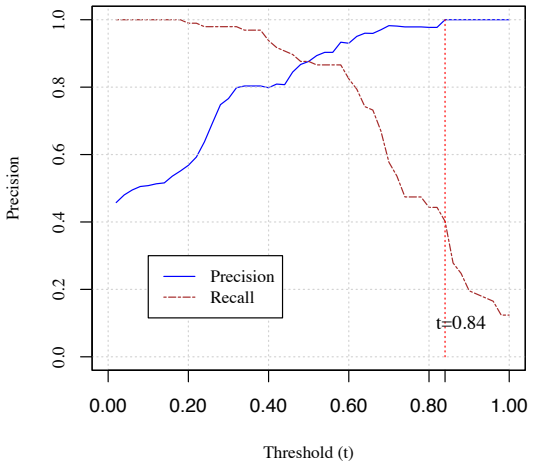


Figure 4: Precision-recall analysis of the behavior deviation threshold value t. Threshold value $t = 0.84$ gives the highest recall rate with 100% detection precision.

the hierarchical similarity-based deviation score with respect to different values for deviation threshold t. We can see that with the threshold value $t = 0.70$, more than half of the evasive samples can be detected with above 98% precision. While with threshold $t = 0.84$, we get 100% precision with the recall rate of 40.20%. In the next large-scale experiment, we used this threshold value $t = 0.84$ to have a high confidence on the detection results.

5.2 Experiment II

In this experiment, we applied our behavior comparison approach to the incoming malware feed received by Anubis. We first filtered the incoming samples based on the size and the type of the behavioral profile extracted in the Anubis analysis environment. This is required to select *interesting* samples from the large volume of the incoming malware without introducing any strong bias. This *pre-selection* process randomly selects samples from the following four groups.

- Samples with minimal activity: These are the samples that show minimal to no activity in the Anubis analysis environment (less than 1000 events). This group may contain evasive malware that successfully evade Anubis analysis environment.

- Samples with high system and network activity: These are the samples that show a substantial amount of system-related activity (more than 1000 events) and network activity in the Anubis analysis environment. We include this group to see if similar behavior can be observed in all other analysis environments.

- Samples with high network activity: These are the samples that show minimal system activity but high network activity (more than 10 packets) in the Anubis analysis environment.

- Samples with high system activity: These are the samples that show no network activity but high system activity in the Anubis analysis environment.

We selected 110,005 samples from the above groups observed during a four months period, starting from July 2013. We extracted behavioral profiles of these samples from all four analysis environments and computed the deviation scores. We used the behavior deviation threshold of $t = 0.84$ that was selected in the previous experiment. With this threshold, we were able to detected 5,835 samples as evasive. That is, these evasive samples evade one or more analysis environments. Since we do not have the ground truth for this dataset, we cannot provide the true precision of this detection.

We also categorized these evasive samples into environment-specific evasions by using the number of events observed within the various analysis environments. That is, if no activity that results in persistent change is observed in an analysis environment a_x while the same sample shows persistent change activities in at least one of the other analysis environments $[a_1, ..., a_n]$, we considered that as an evasion of the analysis environment a_x. The results are presented in Table 2. We found that 2,530 samples showed no activity that resulted in persistent change in all three common malware analysis environments, while such persistent change activities were observed in the bare-metal analysis environment. We see that many samples were able to evade the Ether environment, which is considered as a transparent environment. Manual analysis of those samples revealed that many samples were able to fingerprint Ether by leveraging the dependency of the Xen-hypervisor on Qemu-based *device models* to emulate required virtual devices, such as hard disks. Since Ether is based on Xen-hypervisor, it also requires the Qemu-based device models to run the unmodified guest, which acts as the transparent analysis host. Therefore, if a malware sample inside Ether tries to fingerprint its devices, it might identify Ether as a Qemu-based analysis system. In addition, system call tracing in Ether introduces additional overhead to the system, and, in some of the cases, regular timing-based attacks were successful in detecting the presence of the analysis system.

Table 2: Evasion detection on different environments

Environment	Detection count	Percentage
Anubis	4,947	84.78
Ether	4,562	78.18
VirtualBox	3,576	61.28
All	2,530	43.35
Total	5,835	

6 Limitations

In this section, we discuss the limitations of our approach.

One of the limitations inherent to all dynamic analysis systems is the inability of handling stalling code in a general way. A malware sample can wait for some time (e.g., tens of minutes) before performing any malicious activity. Kolbitsch et al. have proposed a technique to detect and mitigate malware stalling code [26]. However, the mitigation technique is only possible where fine-grained analysis is possible. The technique is not applicable in our bare-metal system. A more sophisticated stalling attack may encrypt the malicious payload with some weak encryption key, which the malware will brute-force during the execution. It is difficult to expedite such process to reveal the malicious activities within a limited execution time. To this end, BareCloud will not be able to observe activity in malware with stalling code.

Malware may employ heuristic checks for anomalies in the internal or external environments that are specific to malware analysis systems. For example, it can wait for real user activities, or try to resolve known non-existent domain using a public DNS server. If there is no user activity in the system or if every non-existent domain is resolved, it is likely that the system is an analysis environment. One of the mitigation techniques can be to remove external anomalies by not actively intercepting and modifying the network communication. User activities can be injected into the system by using automation techniques. However, transparently performing such automation in a bare-metal system is challenging. To this end, such heuristic checks will be able to detect our bare-metal analysis system.

Although, our similarity measure is less sensitive to randomization introduced by the non-deterministic code of a malware sample, high levels of randomization in the persistent changes will result in an erroneous detection of evasion. Other mitigation techniques, such as the multiple execution of the malware on the same analysis system, can be performed to mitigate this problem at the expense of more computational resources.

Both known and unknown fingerprinting techniques focused on detecting virtualized or emulated platforms will fail to detect BareCloud, because we are executing malware on a bare-metal hardware. However, there is the possibility that our system can be fingerprinted by examining unique software/hardware environment features, such as the MAC address of the network device or the presence of the iSCSI driver. In the case of emulated/virtualized environments, it is trivial to randomize such information for each malware analysis run. Since our system uses real hardware, introducing this randomization while preserving the transparency is difficult. The iSCSI driver detection can be mitigated by using more expensive hardware iSCSI initiator instead of a software iSCSI initiator. A hardware iSCSI initiator is a host bus adapter (HBA) that appears to the OS as a hardware storage device. To this end, our system runs as a private malware analysis framework and all outside network communications are blocked. A limited access to the Internet is provided through proxy. As long as the unique environment variables are not leaked to the malware authors, the system can be kept undetectable. However, a dedicated attacker may detect any dynamic analysis system that allows external network communications like ours by using active reconnaissance-based attacks [37]. Malware writers can upload decoy samples to public malware analysis systems so that it is eventually picked up by private analysis systems, such as ours. Such samples can leak unique environment artifacts of these analysis systems using "phoning home" technique and can be used for active reconnaissance.

7 Related works

7.1 Dynamic analysis

Researchers have developed many dynamic analysis tools to analyze malware. These tools mostly focus on extracting system call or Windows API call traces. Many of these analysis systems are based on sandboxing techniques [1, 4, 14, 35]. A sandbox is an instrumented execution environment that executes the malware sample in a contained way. Some of these sandboxes leverage in-guest techniques to trace Windows API calls, such as CWSandbox [35] and Norman Sandbox [4]. Other sandbox systems are implemented using emulation or virtualization technologies. VMScope [22], TTAnalyze [11], and Panorama [36] are some of the examples of emulation-based malware analysis systems. All of them are based on Qemu [12] and implement whole-system emulation. Other tools, such as Ether [14] and HyperDBG [15] are based on hardware-supported virtualization technology. While most system deal with user-land malware samples, some of the analysis systems are specifically targeted to analyze kernel-mode malware [27, 30].

7.2 Transparent analysis

Many transparent malware analysis systems have been proposed to defeat evasive malware. Cobra [34] was the first analysis system specifically focused on defeating anti-debugging techniques. However, Cobra runs its tool at the same privilege level as the malware. In principle, this approach makes it impossible to provide absolute transparency.

Many of the malware analysis tools based on the out-of-VM approach are designed to provide better transparency [1, 14, 22], as the analysis system is completely external to the execution environment. However, detection techniques have been developed to detect these analysis systems as well. There are several techniques to detect VMWare [9, 18, 33], as well as Bochs and Qemu [9, 18, 31]. Pek et al. [32] have shown that hardware virtualization-based Ether [14] can be detected using local timing attacks.

The most effective way to provide transparency is to run on real hardware, with an environment that has not been extended with analysis artifacts. BareBox [25] and Nvmtracer [5] both provide bare-metal environments for malware analysis.

7.3 Evasion detection

Chen et al. proposed a detailed taxonomy of evasion techniques used by malware against dynamic analysis system [13]. Lau et al. have employed a dynamic-static tracing technique to detect VM detection techniques. Kang et al. [24] proposed a scalable trace-matching algorithm to locate the point of execution diversion between two executions. The system is able to dynamically modify the execution of the whole-system emulator to defeat anti-emulation checks. Balzarotti et al. [9] proposed a system for detecting dynamic behavior deviation of malware by comparing behaviors between an instrumented environment and a reference host. The comparison method is based on the deterministic program execution replay. That is, the malware under analysis is first executed in a reference host while recording the interaction of the malware with the operating system. Later, the execution is replayed deterministically in a different analysis environment by providing system call return value recorded in the first run, in the assumption that any deviation in the execution is evidence of some kind of environment fingerprinting. Disarm [28] compares behavioral profiles of four emulation-based analysis environments. The behavior comparison requires each sample to be analyzed multiple times in each analysis environment. The main difference between Disarm and our work is that our analysis systems are based on four fundamentally different analysis platforms, including the transparent bare-metal environment with no monitoring component present in the hardware. Moreover, we propose an improved behavior comparison technique that captures the inherent similarity hierarchy of the behavior features, and do not require the resource-expensive execution of same sample multiple times in the same analysis environment.

7.4 Hierarchical Similarity

Hierarchies are used to encode domain knowledge about different levels of abstraction in the type of events observed. They have been used in different field of similarity detection, such as finding text similarity [16], detecting association rules using hierarchies of concepts [21], and finding similarity among deformable shapes [17]. Ganesan et al. [19] proposed a similarity measure that incorporates hierarchical domain structure. However, the similarity computation is focused on the element-level similarity rather than the profile-level similarity. It uses a modified version of cosine-similarity measure.

8 Conclusions

Dynamic analysis is an effective approach for analyzing and detecting malware that uses advanced packing and obfuscation techniques. However, evasive malware can fingerprint such analysis systems, and, as a result, stop the execution of any malicious activities. Most of the fingerprinting techniques exploit the fact that dynamic analysis systems are based on virtualized or emulated environments, which can be detected by several known methods. The ultimate way to thwart such detection is to analyze malware in a bare-metal environment.

In this work, we presented BareCloud, a system for automatically detecting evasive malware by using hierarchical similarity-based behavioral profile comparison. The profiles are collected by running a malware sample in bare-metal, virtualized, emulated, and hypervisor-based analysis environments.

Future work will focus on improving the transparency of the bare-metal analysis component and on developing an iSCSI module that can extract high-level, intermediate file system operation, providing a richer filesystem-level event trace.

9 Acknowledgments

This work is supported by the Office of Naval Research (ONR) under grant N00014-09-1-1042 and the Army Research Office (ARO) under grant W911NF-09-1-0553.

References

[1] Anubis. *http://anubis.iseclab.org*.

[2] Cuckoo Sandbox. *http://www.cuckoosandbox.org*.

[3] Intel Virtualization Technology. *http://intel.com/technology/virtualization*.

[4] Norman Sandbox. *http://www.norman.com/*.

[5] Nvmtrace. *http://code.google.com/p/nvmtrace*.

[6] SleuthKit. *http://www.sleuthkit.org*.

[7] VirtualBox. *http://www.virtualbox.org*.

[8] BAILEY, M., OBERHEIDE, J., ANDERSEN, J., MAO, Z. M., JAHANIAN, F., AND NAZARIO, J. Automated Classification and Analysis of Internet Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2007).

[9] BALZAROTTI, D., COVA, M., KARLBERGER, C., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Efficient Detection of Split Personalities in Malware. In *Symposium on Network and Distributed System Security (NDSS)* (February 2010).

[10] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, Behavior-Based Malware Clustering. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[11] BAYER, U., KRUEGEL, C., AND KIRDA, E. TTAnalyze : A Tool for Analyzing Malware. *European Institute for Computer Antivirus Research (EICAR)* (2006).

[12] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005).

[13] CHEN, X., ANDERSEN, J., MAO, Z. M., BAILEY, M., AND NAZARIO, J. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *IEEE Conference on Dependable Systems and Networks With FTCS and DCC* (2008), IEEE.

[14] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security (CCS)* (2008).

[15] FATTORI, A., PALEARI, R., MARTIGNONI, L., AND MONGA, M. Dynamic and Transparent Analysis of Commodity Production Systems. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2010), ACM.

[16] FELDMAN, R., AND DAGAN, I. Knowledge Discovery in Textual Databases (KDT). In *Conference on Knowledge Discovery and Data Mining (KDD)* (1995).

[17] FELZENSZWALB, P. F., AND SCHWARTZ, J. D. Hierarchical Matching of Deformable Shapes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2007), IEEE.

[18] FERRIE, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[19] GANESAN, P., GARCIA-MOLINA, H., AND WIDOM, J. Exploiting Hierarchical Domain Structure to Compute Similarity. *ACM Transactions on Information Systems (TOIS)* (2003).

[20] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility Is Not Transparency: VMM Detection Myths and Realities. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2007).

[21] HAN, J., AND FU, Y. Discovery of Multiple-level Association Rules from Large Databases. In *Conference on Very Large Data Bases (VLDB)* (1995).

[22] JIANG, X., AND WANG, X. Out-of-the-Box Monitoring of VM-Based High-Interaction Honeypots. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2007).

[23] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *IEEE Symposium on Security and Privacy* (2011).

[24] KANG, M., YIN, H., AND HANNA, S. Emulating Emulation-resistant Malware. *ACM Workshop on Virtual machine security* (2009).

[25] KIRAT, D., VIGNA, G., AND KRUEGEL, C. BareBox : Efficient Malware Analysis on Bare-Metal. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[26] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In

ACM Conference on Computer and Communications Security (CCS) (2011).

[27] LANZI, A., SHARIF, M., AND LEE, W. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[28] LINDORFER, M., KOLBITSCH, C., AND COMPARETTI, P. M. Detecting Environment-Sensitive Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2011).

[29] MANKIN, J., AND KAELI, D. Dione: A Flexible Disk Monitoring and Analysis Framework. *Research in Attacks, Intrusions, and Defenses* (2012).

[30] NEUGSCHWANDTNER, M., PLATZER, C., COMPARETTI, P. M., AND BAYER, U. dAnubis Dynamic Device Driver Analysis Based on Virtual Machine Introspection. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2010).

[31] PALEARI, R., MARTIGNONI, L., ROGLIA, G. F., AND BRUSCHI, D. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *USENIX Workshop on Offensive Technologies (WOOT)* (2009).

[32] PÉK, G., BENCSÁTH, B., AND BUTTYÁN, L. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *European Workshop on System Security (EUROSEC)* (2011), ACM.

[33] RUTKOWSKA, J. Red Pill or how to detect VMM using (almost) one CPU instruction. *http://invisiblethings.org/papers/redpill.html*, 2004.

[34] VASUDEVAN, A., AND YERRABALLI, R. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In *IEEE Symposium on Security and Privacy* (2006).

[35] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy 5*, 2 (Mar. 2007).

[36] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama : Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[37] YOSHIOKA, K., HOSOBUCHI, Y., ORII, T., AND MATSUMOTO, T. Your Sandbox is Blinded: Impact of Decoy Injection to Public Malware Analysis Systems. *Journal of Information Processing* (2011).

Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components

Manuel Egele
Carnegie Mellon University
megele@cmu.edu

Maverick Woo
Carnegie Mellon University
pooh@cmu.edu

Peter Chapman
Carnegie Mellon University
peter@cmu.edu

David Brumley
Carnegie Mellon University
dbrumley@cmu.edu

Abstract

Matching function binaries—the process of identifying similar functions among binary executables—is a challenge that underlies many security applications such as malware analysis and patch-based exploit generation. Recent work tries to establish semantic similarity based on static analysis methods. Unfortunately, these methods do not perform well if the compared binaries are produced by different compiler toolchains or optimization levels. In this work, we propose *blanket execution*, a novel dynamic equivalence testing primitive that achieves complete coverage by overriding the intended program logic. Blanket execution collects the side effects of functions during execution under a controlled randomized environment. Two functions are deemed similar, if their corresponding side effects, as observed under the same environment, are similar too.

We implement our blanket execution technique in a system called BLEX. We evaluate BLEX rigorously against the state of the art binary comparison tool BinDiff. When comparing optimized and un-optimized executables from the popular GNU coreutils package, BLEX outperforms BinDiff by up to 3.5 times in correctly identifying similar functions. BLEX also outperforms BinDiff if the binaries have been compiled by different compilers. Using the functionality in BLEX, we have also built a binary search engine that identifies similar functions across optimization boundaries. Averaged over all indexed functions, our search engine ranks the correct matches among the top ten results 77% of the time.

1 Introduction

Determining the semantic similarity between two pieces of binary code is a central problem in a number of security settings. For example, in automatic patch-based exploit generation, the attacker is given a pre-patch binary and a post-patch binary with the goal of finding the patched vulnerability [4]. In malware analysis, an analyst is given a number of binary malware samples and wants to find similar malicious functionality. For instance, previous work by Bayer et al. achieves this by clustering the recorded execution behavior of each sample [2]. Indeed, the semantic similarity problem is important enough that the DARPA CyberGenome program has spent over $43M to develop new solutions to it and its related problems [7].

An inherent challenge shared by the above applications is the problem of semantic binary differencing ("diffing") between two binaries. A number of binary diffing tools exist, with current state-of-the-art diffing algorithms such as zynamics BinDiff[1] [8, 9] taking a graph-theoretic approach to finding similarities and differences. BinDiff takes as input two binaries, finds functions, and then performs graph isomorphism (GI) detection on pairs of functions between the binaries. BinDiff highlights pairs of function code blocks between the binaries that are similar and different. Although the graph isomorphism problem has no known polynomial time algorithm, BinDiff has been carefully designed with clever heuristics to make it usably fast in practice. This graph-theoretic approach pioneered by BinDiff has inspired follow-up work such as BinHunt [10] and BinSlayer [3].

While GI-based approaches work well when two semantically equivalent binaries have similar control flow graphs (CFG), it is easy to create semantically equivalent binaries that have radically different CFGs. For example, compiling the same source program with `-O0` and `-O3` radically changes both the number of nodes and structure of edges in both the control flow graph and the call graph. Our experiments show that even this common change to the compiler's optimization level invalidates this assumption and reduces the accuracy of the GI-based BinDiff to 25%.

In this paper, we present a new binary diffing algorithm that does not use GI-based methods and as a result finds similarities where current techniques fail. Our insight is that regardless of the optimization and obfuscation differ-

[1] http://www.zynamics.com/bindiff.html

ences, similar code must still have semantically similar execution behavior, whereas different code must behave differently. At a high level, we execute functions of the two input binaries in tandem with the same inputs and compare observed behaviors for similarity. If observed behaviors are similar across many randomly generated inputs, we gain confidence that they are semantically similar. The main idea of executing programs on many random inputs to test for semantic equivalence is inspired by the problem of polynomial identity testing (PIT). At a high level, the PIT problem seeks efficient algorithms to test if an arithmetic circuit C that computes a polynomial $p(x_1, \cdots, x_n)$ over a given base field $\mathbb{F}$ outputs zero for *every* one of the $|\mathbb{F}|^n$ possible inputs. The earliest algorithm for PIT was a very intuitive randomized algorithm that simply runs C on random inputs. This algorithm depends on the fact that if p is not identically zero, then the probability that C returns zero on a randomly-chosen input is "*small*."[2] By repeating this test, either we will hit an input $(x_1, \cdots, x_n)$ such that $p(x_1, \cdots, x_n) \neq 0$, or we gain confidence that p is identically zero.

There are many challenges to applying the general PIT idea to actual programs, however. Arithmetic circuits have well-defined inputs and outputs, but it is currently an area of active research to identify the inputs and outputs of functions in binary code (see, e.g., [5]). Instead, we propose seven assembly-level features to record during the execution of each function as an approximation of its semantics. Additionally, while it is straightforward to evaluate an arithmetic circuit entirely, finding a collection of inputs that can execute and thus extract the semantics of every part of a program is another open research problem. To achieve full coverage, we repeatedly start execution from the first un-executed instruction of a function until every instruction has been executed at least once.

We have implemented a dynamic equivalence testing system called BLEX to evaluate our blanket execution technique. Our system observes seven semantic features from an execution, namely the four groups of values read from and written to the program heap and stack, the calls made to imported library functions, the system calls made during execution, and the values stored in the `%rax` register upon completion of the analyzed function. We compute the semantic similarity of two functions by taking a weighted sum of the Jaccards of the seven features. Our evaluation is based on a comprehensive dataset. Specifically, we compiled GNU coreutils 8.13 with three current compiler toolchains—`gcc` 4.7.2, `icc` 14.0.0, and `clang` 3.0-6.2. Then, for each compiler toolchain, we compiled coreutils at optimization levels 0 to 3, producing 12 versions of coreutils in total.

[2]The precise upperbound on this probability is commonly known as the Schwartz-Zippel Lemma [23, 28].

Overall, our contributions are as follows:

- We propose *blanket execution*, a novel full-coverage dynamic analysis primitive designed to support semantic feature extraction (§3). Unlike previous approaches such as [25], blanket execution ensures the execution of every instruction without forced violation of branch instruction semantics.
- We propose seven binary code semantics extractors for use in blanket execution. This allows us to approximate the semantics of a function without relying on variable identification or source code access.
- We implement the proposed algorithm in a system called BLEX and evaluate it on a comprehensive dataset based on GNU Coreutils compiled on 4 optimization levels by 3 compilers. Our experiments show that BinDiff performs well (8% better than BLEX) on binaries that are syntactically similar. For binaries that show significant syntactic differences, BLEX outperforms BinDiff by a factor of up to 3.5 and a factor of 2 on average.

2 Problem Setting and Challenges

The problem of matching function binaries is a significant challenge in computer security. In this problem setting, we are only given access to binary code without debug symbols or source. We assume the code is not packed and is compiled from a high-level language that has the notion of a function, i.e., not hand-written assembly. While handling packed code is important, it poses unique challenges which are out of scope for this paper. There are many real-life examples of such problem settings in security. These include, for example, automatic patch-based exploit generation [4], reverse engineering of proprietary code [24], and finding bugs in off-the-shelf software [6].

Clearly, all compiled versions of the same source code should be considered similar by a system addressing the problem of matching function binaries. In this paper, we explicitly consider the case where different compilers and optimization settings produce different binary programs from identical source code. Changing or updating compilers and optimizers happens periodically in industry. For example, with the release of the Xcode 4.1 IDE, Apple switched the default compiler suite from `gcc` to `llvm` [1]. Furthermore, changing compilers and optimization settings is similar to an obfuscation technique. It is common for optimizers to substitute instruction sequences with semantically equivalent but syntactically different sequences. This is exactly a form of metamorphism.

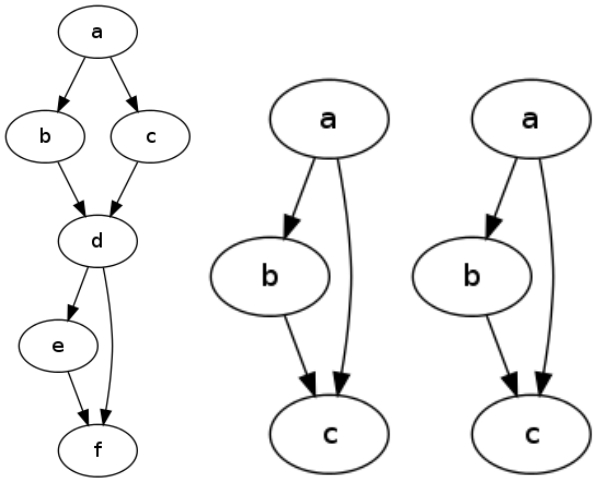
As a motivating example, consider the problem of determining similarities in `ls` compiled with `gcc -O0` and `gcc -O3`, as shown in Figure 1. Although the two assembly listings are the result of the exact same source code, almost all syntactic similarities have been eliminated by the applied compiler optimizations. *If we cannot handle*

```
1   static int strcmp_name(
2       V a, V b
3   )
4   {
5   return cmp_name(a, b, strcmp);
6   }
7
8   static inline int
9   cmp_name (
10    struct fileinfo const *a,
11    struct fileinfo const *b,
12    int (*cmp) (
13       char const *,
14       char const *)
15  )
16  {
17    return
18        cmp (a->name, b->name);
19  }
```

```
1   407ab9 <strcmp_name>:
2      ab9: push %rbp
3   ...
4      ad1: mov $0x402710,%edx
5   ... PLT entry of strcmp
6      ad6: mov %rcx,%rsi
7      ad9: mov %rax,%rdi
8      adc: callq 406fa1 <cmp_name>
9      ae1: leaveq
10     ae2: retq
11
12  406fa1 <cmp_name>:
13     fa1: push %rbp
14  ...
15     fcd: callq *%rax
16  ... call function pointer (e.g., strcmp)
17     fcf: leaveq
18     fd0: retq
```

```
1   4053e0 <strcmp_name>:
2       e0: mov (%rsi),%rsi
3       e3: mov (%rdi),%rdi
4       e6: jmpq 402590 <strcmp@plt>
```

Figure 1: `strcmp_name` from `ls`. Source (left), compiled with `gcc -O0` (center), and `gcc -O3` (right).



(a) md5_finish_ctx (unoptimized) (b) md5_finish_ctx (optimized) (c) xstrxfrm (optimized)

Figure 2: Only the CFG (b), but not (c), is the correct match for (a).

a short function in `coreutils` "obfuscated" only by different optimization levels, what hope do we have on real threats?

The difference between optimized and non-optimized code illustrates several key challenges for correctly identifying the two code sequences as similar:

- Semantically similar functions may not yield syntactically similar binaries. The length of code and operations performed between the two optimization levels is radically different although they both carry out the same simple operation.
- The analysis needs to reason about how memory is read and written. For example, the `-O0` and `-O3` access their arguments identically despite `-O3` not setting up a typical stack frame. In addition, the `cmp_name` function in the `-O0` code up to the call on line 15 indexes struct fields in a semantically equivalent manner to lines 1 and 2 of the `-O3` version.
- Inter-procedural and context sensitive analysis is a must. In `-O0`, `strcmp_name` will always call `cmp_name` with a function pointer pointing to `strcmp`, but in `-O3`, `strcmp` is called directly.

Unfortunately, existing systems both in the security and the general systems community do not address all the above challenges. Syntax-only approaches such as BitShred [12] and others will fail to find any similarities in the code presented in Figure 1. GI-based algorithms will fail because the call and control flow graphs are radically different. GI based methods, such as BinDiff, also face challenges when the control flow graphs to compare are small and collisions render them indistinguishable. Consider, for example the three control flow graphs in Figure 2. The CFG in (a) is the unoptimized version of the `md5_finish_ctx` function in the `sort` utility. While Figure (b) is the optimized version of that function, Figure (c) is the implementation of `xstrxfrm` in the same binary. However, an approach that solely relies on graph similarities, will likely not be able to make a meaningful distinction in this scenario. Alternative approaches, such as the one proposed by Jiang and Su [14] perform only intra-procedural analysis and thus are not able to identify the similarity of the two implementations. To address the above-mentioned challenges in the scope of matching function binaries, we propose a novel dynamic analysis.

3 Approach

We propose *blanket execution* as a novel dynamic analysis primitive for semantic similarity analysis of binary code. Blanket execution of a function f dynamically executes the function repeatedly and ensures that each

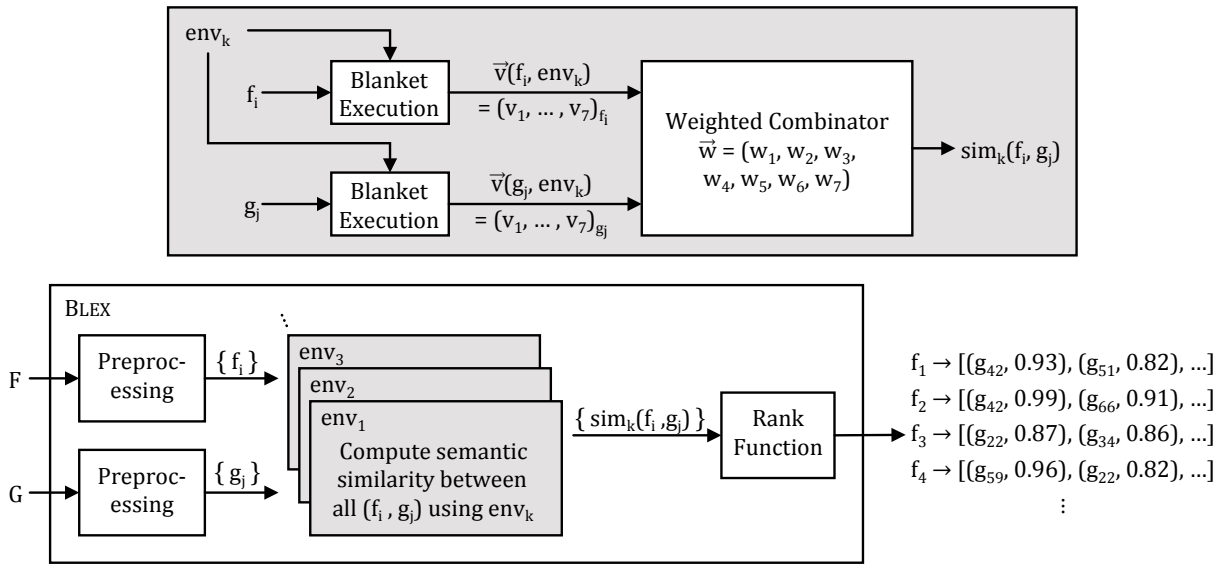


Figure 3: System overview. The upper diagram shows how blanket execution is used to compute the semantic similarity between two given functions f and g inside a given environment env_k. The lower diagram shows how the above computation is used in our BLEX system to, given two program binaries F and G, compute for each function $f_i \in F$ a list of (function, similarity) pairs where each function is a function in G and the list is sorted in non-increasing similarity.

instruction in f is executed at least once. To achieve full coverage, blanket execution starts successive runnings at so-far uncovered instructions. During these repeated runnings, blanket execution monitors and records a variety of dynamic runtime information (i.e., features). Similarity of two functions analyzed by blanket execution is then assessed by the similarity of the corresponding observed features.

More precisely, blanket execution takes a function f and an environment *env* and outputs a vector of dynamic features $\vec{v}(f, env)$ whose coordinates are the feature values captured during the blanket execution. We define the concept of "dynamic feature" broadly to include any information that can be derived from observations made during execution. As an example, we define a feature that corresponds to the set of values read from the heap during a blanket execution.

The novelty of our blanket execution approach lies in (i) *how* the function f is executed for the purpose of feature collection and (ii) *what* features are collected so that they are useful for semantic similarity comparisons. We will first look at (i) while assuming an abstract set of N features in (ii). The latter will be fully specified and explained in §4. For convenience, we will denote each coordinate of a vector $\vec{v}$ as v_i.

3.1 Environment

A key concept in blanket execution is the notion of the *environment* in which a blanket execution occurs. Blanket execution is a dynamic program analysis primitive. This means that in order to analyze a target, blanket execution runs the target and monitors its execution.

To concretely run binary code, we need to provide concrete values of the set of registers and memory locations being read. In blanket execution, we provide concrete initial values to *all* registers and *all* memory locations regardless of whether they are read or not. For unmapped memory regions, an environment also specifies a randomized but fixed *dummy* memory-page. Together, this set of values is known as "an environment." The most important property of an environment is that it must be *efficiently reproducible* since we need to be able to efficiently use a specific environment for multiple runs. This is particularly crucial due to our need to compare feature vectors collected from different functions.

3.2 Blanket Execution

Definition. Given a function f and an environment *env*, the blanket execution of f in *env* is the repeated runnings of f starting from the first un-executed instruction of f until every instruction in f has been executed at least once. Each one of these repeated runnings is called a *blanket execution run* of f, or "be-run" for short. Since we will be using a fixed environment to perform be-runs on a large number of functions, we also define a *blanket execution campaign* ("be-campaign") to be a set of be-runs using the same environment.

Notice that the description of blanket execution encompasses a notion of regaining control. There are several possible outcomes after we start to run f. For example,

f may terminate, f may trigger a system exception, or f may go into an infinite loop. We explain how to handle these possibilities in §4.2.

input : Function binary f, Environment *env*
output : Feature vector $\vec{v}(f, env)$ of f in *env*
$\mathbb{I} \leftarrow getInstructions(f)$
$fvec \leftarrow emptyVector()$
while $\mathbb{I} \neq \emptyset$ **do**
  $addr \leftarrow minAddr(\mathbb{I})$
  $(covered, v) \leftarrow be\text{-}run(f, addr, env)$
  $\mathbb{I} \leftarrow \mathbb{I} \setminus covered$
  $fvec \leftarrow pointwiseUnion(fvec, v)$
end
return $fvec$

Algorithm 1: Blanket Execution.

Algorithm. Algorithm 1 outlines the process of blanket execution for a given function f and an execution environment *env*. First, the function f is dissected into the set of its constituent instructions ($\mathbb{I}$). The system executes the function in the environment *env* at the instruction with the lowest address in $\mathbb{I}$, recording the targeted observations. Executed instructions are removed from $\mathbb{I}$ and the process repeats until all instructions of the function have been executed (i.e., $|\mathbb{I}| = 0$). All recorded feature values, such as memory accesses and system call invocations, are aggregated into a feature vector (*fvec*) associated with the function. Each element in the resulting feature vector is the union of all observed effects for the respective feature.

Rationale. A common weakness of dynamic analysis solutions is potentially low coverage of the program under test. Intuitively, this is because a dynamic analysis must provide an input to drive the execution of the program but by definition a fixed input can exercise only a fixed portion of the program. Although multiple inputs can be used in an attempt to boost coverage, it remains an open research problem to generate inputs to boost coverage effectively. Blanket execution side-steps this challenge and attains full coverage by sacrificing the natural meaning of "executing a function," namely executing from the start of it.

3.3 Assessing Semantic Similarity

The output of a blanket execution on a function f in an environment *env* is a length-N feature vector $\vec{v}(f, env)$. In this section we define how to compute $sim_k(f, g)$, the semantic similarity of two functions f and g given two feature vectors $\vec{v}(f, env)$ and $\vec{v}(g, env)$ that were extracted using blanket execution under the same environment env_k.

Definition. All our features are *sets* of values and we use the Jaccard index to measure the similarity between sets. We define $sim_k(f, g)$ to be a normalized weighted sum of the Jaccard indices on each of the N features in env_k. Mathematically, given N weights $w_1, \ldots, w_N$, we define

$$sim_k(f,g) = \sum_{i=1}^{N} \left(w_i \times \frac{|v_i(f, env_k) \cap v_i(g, env_k)|}{|v_i(f, env_k) \cup v_i(g, env_k)|} \right) / \sum_{\ell=1}^{N} w_\ell.$$

The numerator computes the weighted sum of the Jaccard indices and the denominator computes the normalization constant. The normalization ensures that $sim_k(f, g)$ ranges from 0 to 1, capturing the intuition that it is a similarity measure.

Similarity, Not Equivalence! As explained in §1, our work draws inspiration from the randomized testing algorithm for the polynomial identity testing problem. Strictly speaking, if two functions behave differently in just one environment, we can declare that they are inequivalent. However, in order to make such a judgment, we must have a precise and accurate method to capture the execution behavior of a function. While this is straightforward for arithmetic circuits, it is unsolved for binary code in general. Furthermore, in many applications such as malware analysis, analysts may intend to identify *both* identical and similar functions. This is why we assess the notion of semantic similarity for binary code instead of semantic equivalence.

Weights. Different features may carry different degrees of importance. To allow for this flexibility, we use a weighted sum of the Jaccard indexes. We explain our method to compute the weights (w_ℓ) in §5.1.2.

3.4 Binary Diffing with Blanket Execution

Given the ability to compute the semantic similarity of two functions in a fixed environment, we can perform binary diffing using blanket execution. Figure 3 illustrates the workflow of our system, BLEX.

Preprocessing. Given two binaries F and G, we first preprocess them into their respective sets of constituent functions. We denote these sets as $\{f_i\}$ and $\{g_j\}$ respectively.

Similarity Computation with Multiple Environments. Just as in polynomial identity testing, we will compute the similarity of every pair of (f_i, g_j) in multiple randomized environments $\{env_k\}$. Recall from §3.3 that $sim_k(f_i, g_j)$ is the computed semantic similarity of f_i and g_j in env_k.

Ranking by Averaged Similarity. For each (f_i, g_j), we compute their similarity by averaging over the environments. Let K be the number of environments used. Mathematically, we define

$$sim(f_i, g_j) = \frac{1}{K} \sum_k sim_k(f_i, g_j).$$

Finally, for each function f_i in the given binary F, we output a list of (function, similarity) pairs where each function is an identified function in G and the list is sorted in non-increasing similarity. This completes the process illustrated in Figure 3.

4 Implementation

We implemented the approach proposed in §3 in a system called BLEX. BLEX was implemented and evaluated on Debian GNU/Linux 7.4 (Wheezy) in its amd64 flavor. Because BLEX uses the Pin dynamic instrumentation framework [17] (see §4.2), it is easily portable to other platforms supported by Pin (e.g., Windows or 32-bit Linux).

4.1 Inputs to Blanket Execution

BLEX operates on two inputs. The first input is a program binary F, and the second input is an execution environment under which blanket execution is performed. In a first pre-processing step, BLEX dissects F into individual functions f_i. Subsequently, BLEX applies blanket execution to the f_i as explained in §3. Furthermore, Algorithm 1 uses a static analysis primitive `getInstructions`, which dissects a given function into its constituent instructions.

Reliably identifying function boundaries in binary code is an open research challenge and a comprehensive solution to the function boundary identification problem is outside the scope of this work. However, heuristic approaches, such as Rosenblum et al. [22] or the techniques implemented in IDA Pro [11] deliver reasonable accuracy when identifying function boundaries. IDA Pro supports both primitives used by blanket execution (i.e., function boundary identification and instruction extraction). BLEX thus defers these tasks to the IDA Pro disassembler.

4.2 Performing a BE-Run

A blanket execution run starts execution of a function at a given address *i* under an environment *env*. However, given a program binary, one cannot just instruct the operating system to start execution at said address. Upon program startup, the operating system and loader are responsible for mapping the executable image into memory and transferring control to the program entry point defined in the file header. We leverage this insight to correctly load the application into memory. Once the loader transfers control to the program entry point, we divert control to the address from which to perform the blanket execution run (address *i*). Letting the loader perform its intended operation means that the executable will be loaded with its valid expected memory layout. Note that valid here only means that all sections of the binary are correctly mapped to memory.

Applications frequently make use of functions imported from shared libraries. On Linux the runtime linker implements lazy evaluation of entries in the procedure linkage table (plt). That is, function addresses are only resolved the first time the function is called. However, the side effects produced by the dynamic linker are not characteristic of function behavior. Instead, these side effects create unnecessary noise during blanket execution. To prevent such noise, BLEX sets the `LD_BIND_NOW` environment variable to instruct ld.so (on Linux) to resolve all plt entries at program startup.

Once the be-run starts, BLEX records the side effects produced by the code under analysis. To this end, BLEX leverages the Pin dynamic instrumentation framework to monitor memory accesses and other dynamic execution characteristics, such as system calls and return values (see §4.3). Program code that executes in a random environment is expected to reference unmapped memory. Such invalid memory accesses commonly cause a segmentation fault. To prevent this common failure scenario, BLEX intercepts accesses to unmapped memory. Instead of terminating the analysis, BLEX replaces the referenced (unmapped) memory page with the contents of a dummy memory page specified in the environment. This allows execution to continue without terminating the analysis.

When Does a Run Terminate? A be-run is an interprocedural dynamic analysis of binary code. However, such a dynamic analysis is not guaranteed to always terminate within a reasonable amount of time. In particular, executing under a randomized environment can easily cause a situation where the program gets stuck in an infinite loop. To avoid such situations and guarantee forward progress, BLEX continuously evaluates the following criteria to determine if a be-run is completed.

1. Execution reaches the end of the function in which the be-run started.
2. An exception is raised or a terminal signal is received.
3. A configurable number of instructions have been executed.
4. A configurable timeout has expired.

BLEX detects that a function finished execution by keeping a counter that corresponds to the depth of the call stack. Upon program startup the counter is initialized to zero. Each `call` instruction increments the counter and each `ret` instruction decrements the counter by one. As soon as the counter drops below zero, the be-run is said to be completed.

To catch exceptions and signals, BLEX registers a signal handler and recognizes the end of a be-run if a signal is received. If the code under analysis registered a signal handler for the received signal itself, BLEX does not terminate the be-run but passes the signal on to the appropriate signal handler.

4.3 Instrumentation

Blanket execution monitors the side effects of program execution. A wealth of systems such as debuggers, emulators, and virtual machines have been used in the past to implement dynamic analyses. We chose to implement BLEX on Intel's Pin dynamic instrumentation framework because the tool is mature, well documented, and proven in practice.

At its core, Pin employs a just-in-time (JIT) compiler to recompile basic blocks of binary application code at runtime. Instead of running the original application code, Pin recompiles a block of code, inserting the instrumentation functionality specified by the developer. Then the recompiled code is executed. Upon completion of the code block, Pin regains control and repeats the same process for the next block of application code. The analysis functionality is specified by the developer in a so-called pintool. A pintool is a collection of instrumentation and analysis routines written in C++.

BLEX uses a pintool to instrument individual instructions and record their effects during a be-run. In our implementation we chose features that capture a variety of system level information (e.g., memory accesses), as well as higher level attributes, such as function and system calls. While the current list of features can easily be extended, the following features proved useful for establishing function binary similarity:

1. Values read from the program heap (v_1)
2. Values written to the program heap (v_2)
3. Values read from the program stack (v_3)
4. Values written to the program stack (v_4)
5. Calls to imported library functions via the plt (v_5)
6. System calls made during execution (v_6)
7. Return values stored in the `%rax` register upon completion of the analyzed function (v_7)

Each be-run results in seven sets of observations – one set for each feature. Once all instructions for a function f have been covered, BLEX combines all information pertaining to f in seven sets. That is, given a function f, all observations of v_1 are combined into a single set of values for that function (i.e., f_{v_1}). The same process is repeated for the remaining categories to produce $f_{v_2} \dots f_{v_7}$. The result after blanket execution of a program is the list of functions f_i and the seven sets of observed side effects for each f_i.

Categories $v_1 \dots v_4$ and v_7 record the numeric values used in the respective operations (e.g., values read from memory). Category v_5 records the names of the invoked functions, and v_6 records the system calls invoked. Note that the technique of blanket execution neither defines nor restricts extracted features. BLEX can easily be extended with additional features that help characterize functions. §5 shows that the seven categories of features currently extracted by BLEX are well suited to capture the semantic information of functions.

BLEX relies on the observation that many execution side effects are characteristic of function semantics and thus persist between different compilers and optimizations. While compilers certainly cannot optimize system calls without sacrificing correctness, memory accesses are commonly subject to optimizations. For example, an optimized register allocation scheme can prevent the need for aggressively spilling registers onto the stack. This means that some features are more robust and thus more indicative of function semantics than others. To address these varying degrees of influence, BLEX attributes each feature category with a weight factor ($w_1 \dots w_7$). BLEX leverages support vector machines to establish optimal values for these weights (§5.1.2). We will now discuss how BLEX monitors program execution for side effects.

Memory Accesses. The first four categories of side effects ($v_1 \dots v_4$) are derived from memory accesses. BLEX conforms to a standard memory model where reading a previously written memory cell returns the most recent value written to that cell. While this model is intuitive it only applies to valid (i.e., mapped) memory. In the case that a program tries to access unmapped memory, the operating system will raise an invalid memory reference exception. If a function makes use of global variables, or expects a pointer to mapped memory as one of its formal arguments, normal program execution will initialize such memory properly before the function is called. However, because blanket execution is oblivious to such semantic dependencies, it is common that functions access unmapped memory during blanket execution.

To prevent program termination due to unmapped memory accesses, BLEX simulates that all memory references are valid. To this end, whenever the program tries to access unmapped memory, BLEX interrupts program execution and maps a dummy page in that location. This page is then populated with the data from the dummy page specified in the execution environment.

Pin makes it easy for the developer to emulate data transfers from memory to a register. Thus, a naïve but ineffective approach to simulate that all memory is mapped would be to instrument all instructions that transfer data from memory to a register. For example, the instruction

```
mov (%rsi),%rdx
```

will copy the value pointed to by`%rsi` into register `%rdx`. Unfortunately, Pin's capabilities of intercepting memory accesses are restricted to explicit data transfers such as the above. Instructions with input and output operands that are memory cells cannot be instrumented in the same way. For example, the instruction

```
addl $0x1, $0x20(%rax)
```

will add the constant value one to the value that is stored at offset 0x20 from the memory address in `%rax`. Because Pin's instrumentation capabilities are not fine-grained enough to modify the values retrieved during operand resolution, the straight-forward approach to emulate memory accesses is not generally applicable.

Of course, BLEX needs to collect observations from all instructions that access memory and not just those that explicitly transfer values from memory to the register file or vice versa. Thus, BLEX implements the following mechanisms depending on whether an instruction reads, writes, or reads and writes memory.

Read Accesses. The Pin API allows us to selectively instrument instructions that read from memory. Furthermore, Pin calculates the effective address that is used for each memory accessing instruction. Thus, before an instruction reads from memory, BLEX will verify that the effective address that will be accessed by the instruction belongs to a mapped memory page. If no page is mapped at that address, BLEX will map a valid dummy memory page at the corresponding address[3] and the memory access will succeed.

Recall that a blanket execution environment consists of register values and a memory page worth of data that is kept consistent across all blanket execution runs for a given campaign. By seeding dummy pages with the contents specified in the environment, functions that access unmapped memory will read a consistent value. The rationale is that binary code calculates memory addresses either from arguments or global symbols. Similar functions are expected to perform the same arithmetic operations on these values to derive the memory address to access. Consider, for example, the binary implementations illustrated in Figure 1. Both implementations of `strcmp_name` expect and dereference two pointers to `fileinfo` structures (passed in `%rsi` and `%rdi`). During blanket execution these arguments contain random but consistent values as determined by the execution environment. Dereferencing these random values will likely result in a memory access to unmapped memory. By mapping the dummy page at the unmapped memory region, BLEX ensures that both implementations retrieve the same random value from the dummy page.

With this mechanism in place, BLEX can monitor all read accesses to memory by first making sure that the target memory page is mapped, and then read the original value stored at the effective address in memory.

Write Accesses. Similar to read accesses Pin provides mechanisms to instrument instructions that write to memory. However, the Pin API is not expressive enough to record the values that are written to memory. Thus, to record values that are written to memory, BLEX reads the value from memory after the instruction executed. Similar to the read access technique mentioned above, BLEX will make sure that memory writes succeed by mapping a dummy page at the target address if that address resides in unmapped memory.

Memory Exceptions. BLEX only creates dummy pages for memory accesses to otherwise unmapped memory ranges. If the program tries to access mapped memory in a way that is incompatible with the memory's page protection settings BLEX does not intervene and the operating system raises a segmentation fault. This would occur, for example, if an instruction tries to write to the read-only `.text` section of the program image.

System Calls. Besides memory accesses BLEX also considers the invocation of system calls as side effects of program execution. Pin provides the necessary functionality to intercept and record system calls before they are invoked.

Library Calls. System calls are a well defined interface between kernel space and user space. Thus, they present a natural vantage point to monitor execution for side effects. However, many functions (39% in our experiments) do not result in system calls and thus, relying solely on system calls to identify similar functions is insufficient. Therefore, BLEX also monitors what library functions an application invokes. To support dynamic linking, ELF files contain a `.plt` (procedure linkage table) section. The .plt section contains information (i.e., one entry per function) about shared functions that might be called by the application at runtime. While stripped binaries are devoid of symbol names, they still contain the names of library function names in the plt entries. BLEX records the names of all functions that are invoked via the plt.

While there is no alternative for a program to making system calls, it is not mandatory that shared library functions are invoked through the plt. For example, a developer can chose to statically link a library into her application or interface with the dynamic linker and loader directly by means of the `dlopen` and `dlsym` APIs. Thus, functions from a statically linked version of a program and those from a dynamically linked version thereof will differ in the side effects observed for category library function calls (i.e., v_5).

4.4 Calculating Function Similarity

BLEX combines all of the above methods into a single pintool of 1,036 lines of C++ code. During execution, the pintool collects all necessary information pertaining to the seven observed features. Each be-run results in a feature vector consisting of seven sets that capture the observed side effects. Once all be-runs for a single function finish, BLEX combines the recorded feature vectors and

[3] More precisely, the dummy page is mapped at the target address rounded down to a page-aligned starting address.

associates this information with the function. Because the individual dimensions in the vectors are sets, BLEX uses the set-union operation to combine the individual feature vectors, one dimension at a time. As discussed in §3.3, BLEX assesses the similarity of two functions f and g by calculating the weighted sum of the Jaccard indices of the seven dimensions in the respective feature vectors. We use the Jaccard index as a measure of similarity, because even semantically equivalent functions can result in slight differences in the observed feature values. For example, the unoptimized version in Figure 1 will write and read the passed arguments to the stack, whereas the optimized version does not contain such code. This different behavior results in slightly different values of the corresponding coordinates in the feature vectors.

5 Evaluation

BLEX is an implementation of the blanket execution approach to perform function similarity testing on binary programs. We evaluate BLEX to answer the following questions:

- Can BLEX recognize the similarity between semantically similar, yet syntactically different implementations of the same function? (§5.3)
- Can BLEX match functions compiled from the same source code but with different compiler toolchains and/or configurations? (§5.4)
- Is BLEX an improvement over the industry standard tool, BinDiff? (§5.4)
- Can BLEX be used as the basis for high-level applications? (§5.5)

We begin our evaluation with an experiment on syntactically different implementations of the `libc` function `ffs`, followed by an evaluation of the effectiveness of BLEX over BinDiff across a large set of programs with different compilers and compiler configurations, finishing with a prototype search engine for binary programs built on BLEX. Before presenting our results, we discuss the dataset, ground truth, and feature weights used in the evaluation.

5.1 Dataset

For this evaluation we compiled a dataset based on the popular coreutils-8.13 suite of programs. This version of the coreutils suite consists of 103 utilities. However, to prevent damage to our analysis environment, we excluded potentially destructive utilities such as `rm` or `dd` from the dataset, reducing the number of utilities from 103 to 95. We used three different compilers (gcc 4.7.2, icc 14.0.0, and clang 3.0-6.2) with four different optimization settings (`-O0`, `-O1`, `-O2`, and `-O3`) each to create 12 versions of the coreutils suite for the x86-64 architecture. In total our dataset consists of 1,140 unique binary applications, comprising 195,560 functions.

Feature	Accuracy
Read from heap (v_1)	40%
Write to heap (v_2)	57%
Read from stack (v_3)	58%
Write to heap (v_4)	53%
Library function invocation (v_5)	17%
System calls (v_6)	39%
Function return value (v_7)	13%

Table 1: Accuracy of individual features.

5.1.1 Ground Truth

Although BLEX does not rely on or use debug symbols, we compiled all binaries with the `-g` debug flag to establish ground truth based on the symbol names. For our problem setting, we strip all binaries before processing them with BLEX or BinDiff.

Function inlining has the effect that the inlined function disappears from the target binary. Interestingly, the linker can have the opposite effect when it sometimes introduces duplicate function implementations. For example, when compiling the `du` utility, the linker will include five identical versions of the `mbuiter_multi_next` function in the application binary. While such behavior could be explained if the compiler performed code locality optimization, this also happens if all optimization is turned off (`-O0`). This observation suggests that optimization is not the reason for this code duplication. Because these duplicates are exactly identical, we have to account for this ambiguity when establishing ground truth. That is, matching any of the duplicate instances of the same function should be treated equal and correct. In our dataset 37 different programs contained duplicates (between two and six copies) of 16 different functions. Based on these observations, we establish ground truth by considering functions equivalent if they share the same function name.

5.1.2 Determining Optimal Weights

Each feature in BLEX has a weight factor associated with it, i.e., $w_\ell | \ell = 1 \ldots 7$. To assess the sensitivity of BLEX to these weights, we performed seven small-scale experiments as a sensitivity analysis of the individual features. In each experiment, we set all but one weight to zero and evaluated the accuracy of the system when matching functions between all coreutils compiled with `gcc` and the `-O2` and `-O3` optimization settings. Table 1 illustrates how well the individual features BLEX collects can be used to assess similarity between functions.

To establish the optimal values for these weights, we leveraged the Weka[4] (version 3.6.9) machine learning toolkit. Weka provides an implementation of the sequen-

[4] http://www.cs.waikato.ac.nz/ml/weka/

tial minimal optimization algorithm [20] to train a support vector machine based on a labeled training dataset. To train a support vector machine, the training dataset must consist of feature values for positive and negative examples. We created the dataset based on our ground truth by first selecting 9,000 functions at random from our pool of functions. For each function f in a binary F we calculated the Jaccard index with its correct match g in binary G, constituting a positively labeled sample. For each positively labeled sample, we created a negatively labeled sample by calculating the Jaccard index with the feature vector of a random function $g' \in G$ such that $g' \neq g$. The support vector machine determined the weights as $w_2 = 2.4979$, $w_6 = 0.8775$, $w_4 = 0.4052$, $w_1 = 0.3846$, $w_3 = 0.3786$, $w_7 = 0.3222$, and $w_5 = 0.1082$. Using these weights in BLEX to evaluate the dataset from the above-mentioned sensitivity analysis improved accuracy to 75%.

5.2 Experimental Setup

We evaluated BLEX on a commodity desktop system equipped with an Intel Core i7-3770 CPU (4 physical cores @ 3.4GHz) running Debian Wheezy. For this evaluation we set the maximum number of instructions t_i to 10,000 instructions and the timeout for a single be-run to three seconds. We performed blanket execution for all 195,560 functions in our dataset under eleven different environments. On average, 1,590,773 be-runs were required to cover all instructions in the dataset for a total of 17,498,507 be-runs. A single be-run took on average 0.28 seconds, an order of magnitude below the timeout threshold we selected. Only 9,756 be-runs were terminated because of this timeout. 604,491 be-runs (3.5%) were terminated because the number of instructions exceeded the chosen threshold of 10,000 instructions. While performing blanket execution on all 1,140 unique binaries in our dataset required approximately 57 CPU days, performing blanket execution on two versions of the `ls` utility can be achieved in 30 CPU minutes. Because the repeated runnings in blanket execution are independent of each other, blanket execution resembles an embarrassingly parallel workload and scales almost linear with the number of available CPU cores.

5.3 Comparing Semantically Equivalent Implementations

BLEX tracks the observable behavior of function executions to identify semantic similarity independent of the source code implementation. To test our design, we acquired two different implementations of the `ffs` function from the `Newlib` and `uclibc` libraries as used in the evaluation of the system built by Ramos et al. [21] to measure function equivalence in C source code. We compiled both sources with `gcc -O2`. The resulting binaries differed significantly: the control flow graph in the `uclibc` implementation consisted of eleven basic blocks and the `Newlib` implementation consisted of just four basic blocks. We ran BLEX on both function binaries in 13 different random environments. After comparing the resulting feature vectors, BLEX reported perfect similarity between the compiled functions. This result illustrates how BLEX and blanket execution can identify function similarity despite completely different source implementations.

5.4 Function Similarity across Compiler Configurations

The ideal function similarity testing system can identify semantically similar functions regardless of the compiler, optimizations, and even obfuscation techniques employed. The task is nontrivial as different compiler options can result in drastically different executables (see Figure 1). A rough measure of these differences is the number of enabled compiler optimizations. Consider, for example, the number of optimizations enabled by the four common optimization levels in `gcc`. The switch `-O0` turns off all optimization, and `-O1` enables a total of 31 different optimization strategies. Additionally, `-O2` enables another 26 settings, and `-O3` finally adds another nine optimizations. We would expect that binaries compiled from the same source with `-O2` and `-O3` optimizations are closest in similarity. Thus, similarity testing should yield better results for such similar implementations then for binaries compiled with `-O0` and `-O3` optimizations.

We leverage our dataset to compare the accuracy of BLEX and BinDiff in identifying similar functions of the same program, built with different compilers and different compilation options.

Comparison with BinDiff. BinDiff is a proprietary software product that maps similar functions in two executables to each other. To this end, BinDiff assigns a signature to each function. Function signatures initially consist of the number of basic blocks, the number of control flow edges between basic blocks, and the number of calls to other functions. BinDiff immediately matches function signatures that are identical and unique. For the remaining functions, BinDiff applies secondary algorithms, including more expensive graph analyses. One such secondary algorithm matches function names from debug symbols. However, our experiments do not leverage debugging symbols as our efforts are focused on the performance on stripped binaries. The data presented in this evaluation was obtained with BinDiff version 4.0.1 and the default configuration.

As Figure 4 illustrates, BinDiff is very proficient in matching functions among the same utility compiled with the very similar `-O2` and `-O3` settings. Although BLEX also performs reasonably well, BinDiff outperforms BLEX on almost all utilities in this comparison.

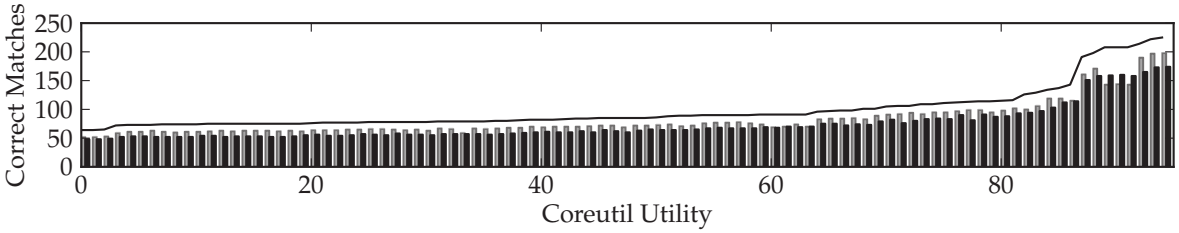


Figure 4: Correctly matched functions for binaries in `coreutils` compiled with `gcc -O2` and `gcc -O3`. BinDiff (grey), BLEX (black), total number of functions in utility (solid line).

The solid line in the figure marks the total number of functions in each utility.

Once the differences between two binaries become more pronounced, BLEX shows considerably improved performance over BinDiff. Figure 5 compares BLEX and BinDiff in identifying similar functions in binaries compiled with the `-O0` and `-O3` optimization settings. This combination of compiler options is expected to produce the least similar binaries and thus should establish a lower bound of the performance one can expect from BLEX and BinDiff respectively. This evaluation shows that BLEX consistently outperforms BinDiff, on average by a factor of two. Furthermore, BLEX matches over three times as many functions correctly for the `du`, `dir`, `vdir`, `ls`, and `chcon` utilities.

Finally, we assess the performance of BLEX and BinDiff on programs built with different compilers. Figure 6 shows the accuracy for binaries compiled with `gcc -O0` and Intel's `icc -O3`. Again, due to the substantial differences in the produced binaries, BLEX consistently outperforms BinDiff in the cross-compiler setting.

Discriminatory power of the similarity score. We also evaluated how well the similarity score tells correct from incorrect matches apart. Similarity scores are normalized to the interval [0,1] with 1 indicating perfect similarity and 0 absolute dissimilarity. In Figure 8, we illustrate the expected similarity value over 10,000 pairs of random functions. On average this expected similarity is 0.12. However, when analyzing the similarity scores of correct matches from the experiment used for Figure 4 (i.e., `gcc -O2` vs. `gcc -O3`), the average similarity score is 0.85. This indicates that the seven features BLEX uses to assess function similarity are indeed suitable to perform this task.

Effects of Multiple Environments. As discussed in §3.4, we proposed to perform blanket execution with multiple environments ($\{env_k\}$). To assess the effects of performing blanket execution under multiple environments, we evaluated how the percentage of correct matches varies as k (the number of environment) increases. Our result is shown in Figure 7. The figure shows that a mild increase (from 50% to 55%) in accuracy up until three environments are used. Interestingly, using more than three environments does *not* significantly improve the accuracy of BLEX. This is in stark contrast to the PIT theory. However, as discussed previously, real-world function binaries are not polynomials and BLEX cannot precisely identify all input and output dependencies of a function. Thus, it may not be surprising that a larger number of random environments does not significantly improve the accuracy of the system. We plan to evaluate alternate strategies for crafting execution environments in a "smart" way in the future.

5.5 BLEX as a Search Engine

Matching function binaries is an important primitive for many higher-level applications. To explore the potential of BLEX as a system building-block, we built a prototype search engine for function binaries. Given a search query (a function f) and a corpus of program binaries, we can use BLEX to find the program most likely to contain an implementation of f. Phrased differently, an analyst presented with an unknown function can search for similar functions encountered in the past. The analyst can then easily apply the knowledge gathered during the previous analysis of similar functions, reducing the time and effort spent on redundant analysis. Similarly, if a match is found in a program for which the analyst has access to debug symbols, the analyst can leverage this valuable information to speed up the analysis of the target function.

To evaluate this application, we chose 1,000 functions at random from the applications compiled with `gcc -O0`. These functions serve as the search queries. We compiled the corpus from programs in coreutils built with `gcc -O1`,`-O2`, and `-O3` respectively (29,015 functions in total). Our prototype search engine ranked the correct match as the first result in 64% of all queries. 77% of the queries were ranked under the first 10 results (e.g., the first page of search results) and 87% were ranked under the first 10 pages of results (i.e., top 100 ranks). Figure 9 depicts this information as the left-hand side of the CDF.

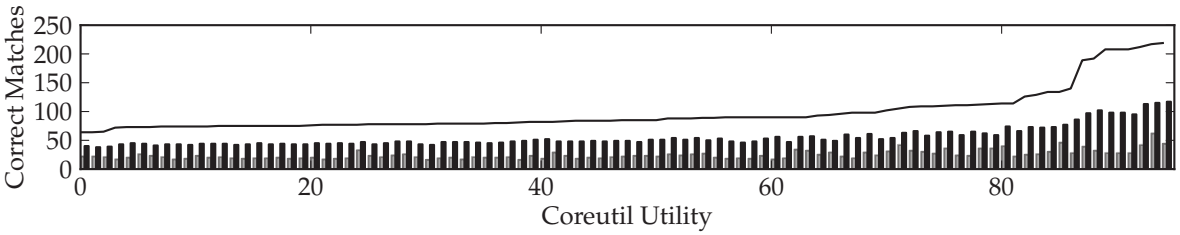


Figure 5: Correctly matched functions for binaries in `coreutils` compiled with `gcc -O0` and `gcc -O3`. BinDiff (grey), BLEX (black), total number of functions in utility (solid line).

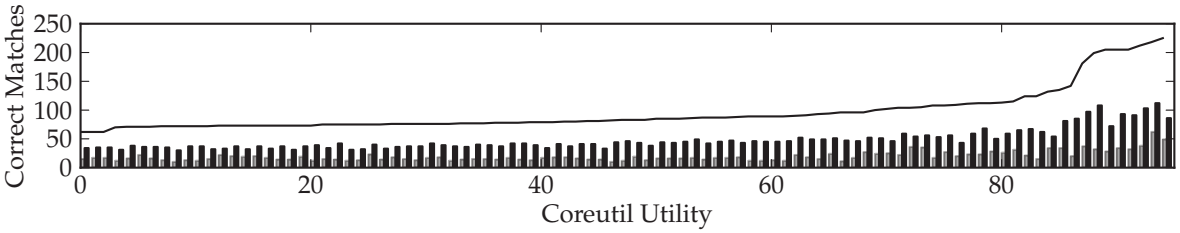


Figure 6: Correctly matched functions for binaries in `coreutils` compiled with `gcc -O0` and `icc -O3`. BinDiff (grey), BLEX (black), total number of functions in utility (solid line).

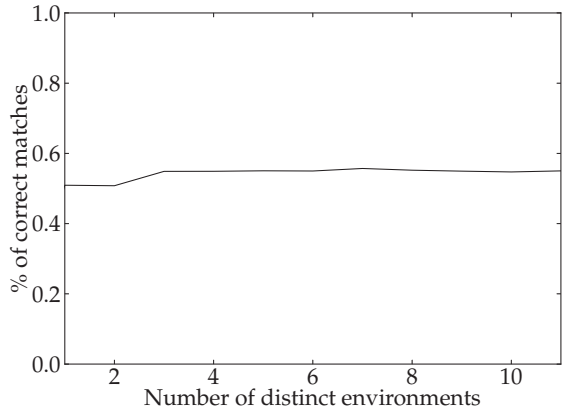


Figure 7: Matching accuracy depending on the number of used environments.

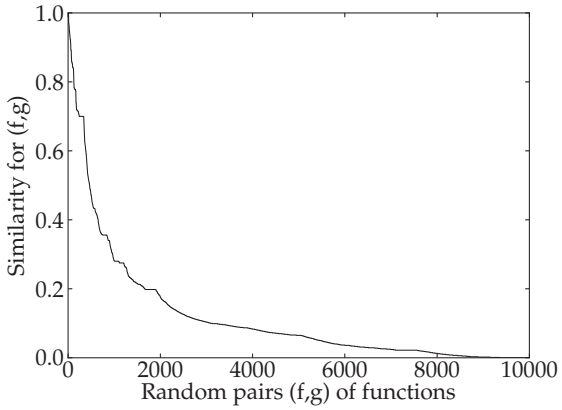


Figure 8: Distribution of similarity scores among 10,000 random pairs of functions. All compiled with `gcc -O2`.

The remaining 13% form a long tail distribution with the worst match at rank 23,261.

The usability of a search engine also depends on its query performance. Our unoptimized implementation answers search queries to the indexed corpus of size 29,015 in under one second on average.

6 Related Work

The problem of testing whether two pieces of syntactically-different code are semantically identical has received much attention by previous researchers. Notably, Jiang and Su [14] recognized the close resemblance of this problem to polynomial identity testing and applied the idea of random testing to automatically mine semantically-equivalent code fragments from a large source codebase. Whereas their definition of semantic equivalence includes only the input-output values of a code fragment and does not consider the intermediate values, we include intermediate values in our features as a pragmatic way to cope with the difficult problem of identifying input-output variables in binary code. Interested readers can see [5, 16] for some of the recent works on that problem.

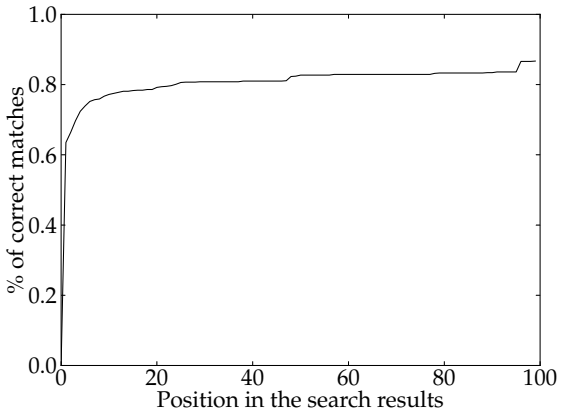


Figure 9: Left-most section of the CDF of ranks for correct matches in 1,000 random search queries.

Intermediate values can also be extremely valuable for other applications. For example, Zhang et al. [26] have investigated how to detect software plagiarism using the dynamic technique of value sequences. This uses the concept of core values proposed by Jhi et al. [13]. The idea is that certain specific intermediate values are unavoidable during the execution of any implementation of an algorithm and are thus good candidates for fingerprinting.

Intermediate values are also used by Zhang and Gupta [27] as a first step in matching the instructions in the dynamic histories of program executions of two program versions. After identifying potential matches as such, Zhang and Gupta refined the match by matching the data dependence structure of the matched instructions. They reported high accuracy in their evaluation using histories from unoptimized and optimized binaries compiled from the same source. This work was used by Nagarajan et al. [18] as the second step of their dynamic control flow matching system. The system by Nagarajan et al. also match functions between unoptimized and optimized binaries. Their technique is based on matching the structure of two dynamic call graphs.

We choose to evaluate BLEX against BinDiff [8, 9] due to its wide availability and also its reputation of being the industry standard for binary diffing. At a high-level, BinDiff starts by recovering the control flow graphs (CFGs) of the two binaries and then attempts to use a heuristic to normalize and match the vertices from the two graphs. Although in essence BinDiff is solving a variant of the graph isomorphism problem of which no efficient polynomial time algorithm is known, the authors of BinDiff have devised a clever neighborhood-growing algorithm that performs extremely well in both correctness and speed if the two binaries are similar. However, as we have explained in the paper, changing the compiler optimization level alone is sufficient to introduce changes that are large enough to confound the BinDiff algorithm.

A noteworthy successor to BinDiff is the BinHunt system introduced in [10]. This paper makes two important contributions. First, it formalized the underlying problem of binary diffing as the Maximum Common Induced Subgraph Isomorphism problem. This allowed the authors to formally and accurately state their backtracking algorithm. Second, instead of relying on heuristics to match vertices and tolerating potential false matches, BinHunt deployed rigorous symbolic execution and theorem proving techniques to *prove* that two basic blocks are in fact equivalent. Unfortunately, BinHunt has only been evaluated in three case studies, all of which support only differences due to patching vulnerabilities. In particular, it has not been evaluated whether BinHunt will perform well on binaries that are compiled with different compiler toolchains or different optimization levels.

A recent addition to this line of work is the BinSlayer system [3]. The authors of BinSlayer correctly observed that graph-isomorphism based algorithms may not perform well when the change between two binaries are large. To alleviate this problem, the authors modeled the binary diffing problem as a bipartite graph matching problem. At a high level, this means assigning a distance between two basic blocks and then pick an assignment (a matching) that maps each basic block from one function to a basic block in another function that *minimizes* the total distance. Among other experiments, the authors evaluated their algorithms by diffing GNU coreutils 6.10 vs. 8.19 (large gap) and also 8.15 vs. 8.19 (small gap). Just as the authors suspected, they observed that graph-isomorphism based algorithms are less accurate in the large-gap experiment than in the small-gap experiment.

Besides binary diffing, our work can also be seen in the light of a binary search engine. Two recent work in the area are Exposé [19] and Rendezvous [15]. Both of these systems are based on static analysis techniques; in contrast, our system is based on dynamic analysis. None of these systems has been evaluated with a dataset that varies both compiler toolchain and optimization level simultaneously.

Finally, semantic similarity can also be used for clustering. For example, Bayer et al. [2] have used ANUBIS for clustering malware based on their recorded behavior. However, this relies on attaining high coverage so that malicious functionality is exposed [25]. We believe that BLEX may also be used for malware clustering.

7 Conclusion

Existing binary diffing systems such as BinDiff approach the challenge of function binary matching from a purely static perspective. It has not been thoroughly evaluated on binaries produced with different compiler toolchains

or optimization levels. Our experiments indicate that its performance drops significantly if different compiler toolchains or aggressive optimization levels are involved.

In this work, we approach the problem of matching function binaries with a dynamic similarity testing system based on the novel technique of blanket execution. BLEX, our implementation of this technique proved to be more resilient against changes in the compiler toolchain and optimization levels than BinDiff.

Acknowledgment

This material is based upon work supported by Lockheed Martin and DARPA under the Cyber Genome Project grant FA975010C0170. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Lockheed Martin or DARPA. This material is further based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 0946825.

References

[1] New Features in Xcode 4.1. https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/WhatsNewXcode/Articles/xcode_4_1.html. Page checked 7/8/2014.

[2] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Network and Distributed System Security Symposium* (2009), The Internet Society.

[3] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM Program Protection and Reverse Engineering Workshop* (2013), ACM.

[4] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), IEEE, pp. 143–157.

[5] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Network and Distributed System Security Symposium* (2010), The Internet Society.

[6] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 380–394.

[7] DARPA-BAA-10-36, Cyber Genome Program. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-36/listing.html. Page checked 7/8/2014.

[8] DULLIEN, T., AND ROLLES, R. Graph-based comparison of executable objects. In *Actes de la Symposium sur la Sécurité des Technologies de l'Information et des Communications* (2005).

[9] FLAKE, H. Structural comparison of executable objects. In *Proceedings of the 2004 Workshop on Detection of Intrusions and Malware & Vulnerability Assessment* (2004), IEEE, pp. 161–173.

[10] GAO, D., REITER, M. K., AND SONG, D. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security* (2008), Springer, pp. 238–255.

[11] HEX-RAYS. The IDA Pro interactive dissassembler. https://hex-rays.com/products/ida/index.shtml.

[12] JANG, J., BRUMLEY, D., AND VENKATARAMAN, S. BitShred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 309–320.

[13] JHI, Y.-C., WANG, X., JIA, X., ZHU, S., LIU, P., AND WU, D. Value-based program characterization and its application to software plagiarism detection. In *Proceeding of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 756–765.

[14] JIANG, L., AND SU, Z. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis* (2009), ACM, pp. 81–92.

[15] KHOO, W. M., MYCROFT, A., AND ANDERSON, R. Rendezvous: A search engine for binary code. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories* (2013), IEEE, pp. 329–338.

[16] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Network and Distributed System Security Symposium* (2011), The Internet Society.

[17] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation* (2005), ACM, pp. 190–200.

[18] NAGARAJAN, V., GUPTA, R., ZHANG, X., MADOU, M., DE SUTTER, B., AND DE BOSSCHERE, K. Matching control flow of program versions. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance* (2007), pp. 84–93.

[19] NG, B. H., AND PRAKASH, A. Exposé: Discovering potential binary code re-use. In *Proceedings of the 37th IEEE Computer Software and Applications Conference* (2013), pp. 492–501.

[20] PLATT, J. C. Sequential Minimal Optimization: A fast algorithm for training Support Vector Machines. Tech. rep., Microsoft Research, 1998.

[21] RAMOS, D. A., AND ENGLER, D. R. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), Springer, pp. 669–685.

[22] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence* (2008), AAAI, pp. 798–804.

[23] SCHWARTZ, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM 27*, 4 (1980), 701–717.

[24] VAN EMMERIK, M. J., AND WADDINGTON, T. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering* (2004), IEEE, pp. 27–36.

[25] WILHELM, J., AND CHIUEH, T.-C. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection* (2007), Springer, pp. 219–235.

[26] ZHANG, F., JHI, Y.-C., WU, D., LIU, P., AND ZHU, S. A first step towards algorithm plagiarism detection. In *Proceedings of 2012 the International Symposium on Software Testing and Analysis* (2012), ACM, pp. 111–121.

[27] ZHANG, X., AND GUPTA, R. Matching execution histories of program versions. In *Proceedings of the 10th European Software Engineering Conference* (2005), ACM, pp. 197–206.

[28] ZIPPEL, R. Probabilistic algorithms for sparse polynomials. In *Proceedings of the 1979 International Symposium on Symbolic and Algebraic Manipulation* (1979), Springer, pp. 216–226.

On the Practical Exploitability of Dual EC in TLS Implementations

Stephen Checkoway[1] Matthew Fredrikson[2] Ruben Niederhagen[3] Adam Everspaugh[2]
Matthew Green[1] Tanja Lange[3] Thomas Ristenpart[2]
Daniel J. Bernstein[3,4] Jake Maskiewicz[5] and Hovav Shacham[5]
[1] *Johns Hopkins University,* [2] *University of Wisconsin,* [3] *Technische Universiteit Eindhoven,*
[4] *University of Illinois at Chicago,* [5] *UC San Diego*

Abstract

This paper analyzes the actual cost of attacking TLS implementations that use NIST's Dual EC pseudorandom number generator, assuming that the attacker generated the constants used in Dual EC. It has been known for several years that an attacker generating these constants and seeing a long enough stretch of Dual EC output bits can predict all future outputs; but TLS does not naturally provide a long enough stretch of output bits, and the cost of an attack turns out to depend heavily on choices made in implementing the RNG and on choices made in implementing other parts of TLS.

Specifically, this paper investigates OpenSSL-FIPS, Windows' SChannel, and the C/C++ and Java versions of the RSA BSAFE library. This paper shows that Dual EC exploitability is fragile, and in particular is stopped by an outright bug in the certified Dual EC implementation in OpenSSL. On the other hand, this paper also shows that Dual EC exploitability benefits from a modification made to the Dual EC standard in 2007; from several attack optimizations introduced here; and from various proposed TLS extensions, one of which is implemented in BSAFE, though disabled in the version we obtained and studied. The paper's attacks are implemented; benchmarked; tested against libraries modified to use new Dual EC constants; and verified to successfully recover TLS plaintext.

1 Introduction

On September 5, 2013, the New York Times [23], the Guardian [3] and ProPublica [16] reported the existence of a secret National Security Agency SIGINT Enabling Project with the mission to "actively [engage] the US and foreign IT industries to covertly influence and/or overtly leverage their commercial products' designs." The revealed source documents describe a US $250 million/year program designed to "make [systems] exploitable through SIGINT collection" by inserting vulnerabilities, collecting target network data, and influencing policies, standards and specifications for commercial public key technologies. Named targets include protocols for "TLS/SSL, https (e.g. webmail), SSH, encrypted chat, VPNs and encrypted VOIP."

The documents also make specific reference to a set of pseudorandom number generator (PRNG) algorithms adopted as part of the National Institute of Standards and Technology (NIST) Special Publication 800-90 [21] in 2006, and also standardized as part of ISO 18031 [15]. These standards include an algorithm called the Dual Elliptic Curve Deterministic Random Bit Generator (Dual EC). As a result of these revelations, NIST reopened the public comment period for SP 800-90.

Known weaknesses in Dual EC. Long before 2013, Dual EC had been identified by the security community as biased [8, 27], extremely slow, and backdoorable.

SP 800-90 had already noted that "elliptic curve arithmetic" makes Dual EC generate "pseudorandom bits more slowly than the other DRBG mechanisms in this Recommendation" [21, p. 177] but had claimed that the Dual EC design "allows for certain performance-enhancing possibilities." In fact, Dual EC with all known optimizations is two orders of magnitude slower than the other PRNGs, because it uses scalar multiplications on an elliptic curve where the other PRNGs use a hash function or cipher call.

The back door is a less obvious issue, first brought to public attention by Shumow and Ferguson [28] in 2007. What Shumow and Ferguson showed was that an attacker specifying Dual EC, and inspecting some Dual EC output bits from an unknown seed, had the power to predict all subsequent output bits.

Specifically, the description of Dual EC standardizes three parameter sets, each specifying an elliptic curve E over a finite field $\mathbf{F}_p$, together with points P and Q on E. The back door is knowledge of $d = \log_Q P$, the discrete logarithm of P to the base Q; an attacker creating P and Q can be assumed to know d. Shumow and Ferguson showed that knowledge of d, together with about $\log_2 p$ consecutive output bits,[1] makes it feasible to predict all subsequent Dual EC output.

Shumow and Ferguson suggested as countermeasures to vary P and Q and to reduce the number of output bits per iteration of the PRNG. However, SP 800-90 requires a particular number of bits per iteration, and states that the standard P and Q "shall be used in applications re-

[*] Date of this document: 2014.06.06.

[1] 256 bits were sufficient in all their P-256 experiments.

Table 1: Summary of our results for Dual EC using NIST P-256.

Library	Default PRNG	Cache Output	Ext. Random	Bytes per Session	Adin Entropy	Attack Complexity	Time (minutes)
BSAFE-C v1.1	✓	✓	✓†	31–60	—	$30 \cdot 2^{15}(C_v + C_f)$	0.04
BSAFE-Java v1.1	✓		✓†	28	—	$2^{31}(C_v + 5C_f)$	63.96
SChannel I‡				28	—	$2^{31}(C_v + 4C_f)$	62.97
SChannel II‡				30	—	$2^{33}(C_v + C_f) + 2^{17}(5C_f)$	182.64
OpenSSL-fixed I*				32	20	$2^{15}(C_v + 3C_f) + 2^{20}(2C_f)$	0.02
OpenSSL-fixed III**				32	$35 + k$	$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$	$2^k \cdot 83.32$

* Assuming process ID and counter known. ** Assuming 15 bits of entropy in process ID, maximum counter of 2^k. See Section 4.3.
† With a library–compile-time flag. ‡ Versions tested: Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2.

The entries in the table are for normal TLS connections. In particular, we exclude all forms of session resumption. A ✓ in the Default PRNG column indicates whether Dual EC is the default PRNG used by the library. A ✓ in the Cache Output column indicates that the unused Dual EC output is cached for use in a subsequent call. A ✓ in the Ext. Random column indicates that the proposed TLS extension Extended Random [25] is supported in some configuration. Reported attack times do not rely on use of Extended Random. Bytes per Session indicates how many contiguous, useful output bytes from Dual EC a TLS server's handshake message reveals. For SChannel II this is an average value of useful bits, see Section 4.2. Adin Entropy indicates how many bits of unknown input are added to each Dual EC generate call. The Attack Complexity is the computational cost in terms of the cost of a scalar multiplication with a variable base point, C_v, and a fixed base point, C_f, in the worst case. With our optimizations (see Section 5), C_f is roughly 20 times faster than C_v; the exact speedup depends on context. The Time column gives our measured worst-case time for the attack on a four-node, quad-socket AMD Opteron 6276 cluster; the time for OpenSSL-fixed III is measured using $k = 0$.

quiring certification under FIPS 140-2"; this stops use of alternative points in certified implementations.

Risk assessment for this back door depends on the probability that the creator of P and Q is an attacker. Shumow and Ferguson wrote "WHAT WE ARE NOT SAYING: NIST intentionally put a back door in this PRNG"; but the September 2013 news indicates that NSA may have deliberately engineered Dual EC with a back door. Our concern in this paper is not with this probability assessment, but rather with impact assessment, especially for the use of Dual EC in TLS.

Use of Dual EC in products. Despite the known weaknesses in Dual EC, several vendors have implemented Dual EC in their products [22]. For example, OpenSSL-FIPS v2 and Microsoft's SChannel include Dual EC, and RSA's crypto libraries use Dual EC by default. RSA Executive Chairman Art Coviello, responding to news that NSA had paid RSA to use Dual EC [18], stated during the opening speech of RSA Conference 2014: "Given that RSA's market for encryption tools was increasingly limited to the US Federal government and organizations selling applications to the federal government, use of this algorithm as a default in many of our toolkits allowed us to meet government certification requirements" [5].

Practical attacks on TLS using Dual EC. This paper studies to which extent *deployed* cryptographic systems that use Dual EC are vulnerable to the back door, assuming that an attacker knows $d = \log_Q P$. Specifically, we perform a case study of Dual EC use in TLS, arguably the most important potential target for attacks. The basic attack described by Shumow and Ferguson [28] (and independently, quietly, by Brown and Vanstone in a patent application [4]) turns out to be highly oversimplified: it does not consider critical limitations and variations in the amount of PRNG output actually exposed in TLS, additional inputs to the PRNG, PRNG reseeding, alignment of PRNG outputs, and outright bugs in Dual EC implementations.

We present not just a theoretical evaluation of TLS vulnerability but an in-depth analysis of Dual EC in four recent implementations of TLS: RSA BSAFE Share for C/C++ (henceforth BSAFE-C), RSA BSAFE Share for Java (henceforth BSAFE-Java), Windows SChannel, and OpenSSL. The Network Security Services (NSS) libraries, e.g., used by Mozilla Firefox, and the TLS implementation of BlackBerry do not offer a Dual EC implementation and thus are not discussed here.

To experimentally verify the actual performance of our attacks, we replace the NIST-specified constants with ones we generate; for BSAFE and Windows this required extensive reverse-engineering of binaries to find not just P and Q but many implementation-specific constants and runtime test vectors derived from P and Q (see Section 4.4). Our major findings are as follows:

- The BSAFE implementations of TLS make the Dual EC back door particularly easy to exploit in two ways. The Java version of BSAFE includes fingerprints in connections, making them easy to identify. The C version of BSAFE allows a drastic speedup in the attack by broadcasting longer contiguous strings of random bits than one would at first imagine to be possible given the TLS standards.

- Windows SChannel does not implement the current Dual EC standard: it omits an important computation. We show that this does not prevent attacks; in fact, it allows slightly faster attacks.
- We discovered in OpenSSL a previously unknown bug that prevented the library from running when Dual EC is enabled. It is still conceivable that someone is using Dual EC in OpenSSL, since the bug has an obvious and very easy fix, so we applied this fix and evaluated the resulting version of OpenSSL, which we call OpenSSL-fixed. OpenSSL-fixed turns out to use additional inputs in a way that under some circumstances makes attacks significantly more expensive than for the other libraries.

When a TLS server uses DSA or ECDSA to sign its DH/ECDH public key, a single known nonce reveals the long-lived signing key which enables future active attacks. Our attacks reveal the inner state of Dual EC which generates the nonces and we have successfully recovered the long-term signing keys.

We also perform a brief measurement study of the IPv4 address space to assess the prevalence of these libraries on the Internet.

We summarize our results in Table 1. The BSAFE-C attack is practically instantaneous, even on an old laptop. The BSAFE-Java and SChannel attacks require more processing power to recover missing bits of Dual EC output. The OpenSSL-fixed attack cost depends fundamentally on how much information on the additional input is available. All of these attacks are practical for a motivated attacker, even when the attacks are repeated against a large number of targets.

2 Dual EC attack theory

Review of Dual EC. We focus on Dual EC using the NIST P-256 elliptic curve. For the curve equation and the standardized base points P and Q see [21, Appendix A.1]. The state of Dual EC is a 32-byte string s, which the user initializes as a secret random seed. The user then calls Dual EC any number of times; each call implicitly reads and writes the state, optionally reads *additional input* from the user, and produces any number of random bytes requested by the user.

Internally, each call works as follows. Additional input, if provided, is hashed and xored into the state. The state is then updated as follows: compute sP and then overwrite s with the x-coordinate $x(sP)$. A 30-byte block of output is then generated as follows: compute sQ, take the x-coordinate $x(sQ)$, and discard the most significant 2 bytes. The resulting 30 bytes are used as output. If more random bytes were requested, the state is updated again and another 30-byte block of output is generated; this repeats until enough blocks have been generated. Any excess bytes in the final block are discarded. The state is updated one final time in preparation for the next call, and the generator returns the requested number of bytes.

Review of the basic attack. The attack from Shumow and Ferguson works as follows. The attacker is assumed to control the initial standardization of P. The attacker takes advantage of this by generating a random secret integer d and generating P as dQ. Alternatively, if the attacker controls the initial standardization of Q rather than P, the attacker generates Q as $(1/d)P$. Either way $P = dQ$, with d secretly known to the attacker.

The idea of the attack is to reconstruct sQ from an output block (see the next paragraph) and then multiply by d, obtaining dsQ, i.e., sP. The x-coordinate $x(sP)$ is the user's next PRNG state. The attacker then computes all subsequent outputs the same way that the user does.

Recall that an output block reveals 30 out of the 32 bytes of the x-coordinate of sQ. The attacker tries all possibilities for the 2 missing bytes, obtaining 2^{16} possibilities for the x-coordinate, and then for each x-coordinate uses the curve equation to reconstruct at most 2 possibilities for the y-coordinate, for a total of at most 2^{17} possibilities for sQ. About half of the x-coordinates will not have any corresponding y-coordinate, and the other half will produce two points (x,y) and $(x,-y)$ that behave identically for the attack, because $x(s(x,y)) = x(s(x,-y))$, so the attacker keeps only one of those points and ends up with about 2^{15} possibilities for sQ. For each of these possibilities, the attacker computes the corresponding possibility for $dsQ = sP$ and for the next Dual EC output. The attacker pinpoints the correct guess by checking the next actual Dual EC output.

Attacks with additional input. Shumow and Ferguson did not analyze the case where a user provides additional input to a Dual EC call. We point out that the analysis of this case depends heavily on whether one considers Dual EC in the June 2006 version of SP 800-90, which we call Dual EC 2006, or Dual EC in the March 2007 version of SP 800-90, which we call Dual EC 2007.

Additional input is only used once at the beginning of a call and therefore does not stop the attacker from using the first 30 bytes of output from a call to predict subsequent output bytes from the same call. The remaining question is whether the attacker can predict the first 30 bytes from a call given the last 30 bytes from the previous call. This issue would be relatively minor if applications were generating many kilobytes of Dual EC output from each call; but the applications we have studied normally generate only one or two blocks from each call, so the predictability of the first 30-byte block is an important question.

If the additional input has enough entropy unknown to the attacker then the answer is no: the first 30 bytes are unpredictable. However, in the applications that we have studied, additional input is either nonexistent or potentially guessable. This is where Dual EC 2006 and Dual EC 2007 produce different answers.

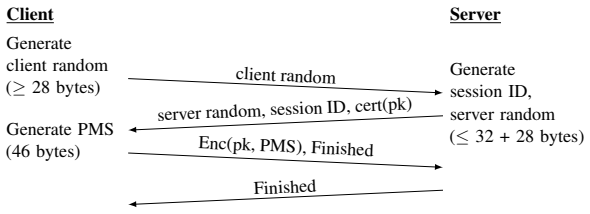
What we have described so far is Dual EC 2007. The previous call returned up to 30 bytes of sQ and produced $s' = x(sP)$ as the new state. Assume that the attacker has reconstructed sQ. This call updates s' to $s'' = x((s' \oplus \mathrm{H}(\mathsf{adin}))P)$, where H is a hash function, and then returns up to 30 bytes of $x(s''Q)$. Given sQ the attacker computes $dsQ = sP$, computes s', and then for each possible adin computes s'' and $s''Q$.

The 2006 version of Dual EC differs slightly from the 2007 version: Dual EC 2006 is missing the final step which updates the seed s at the end of each call. The previous call returned most of sQ but left s untouched. If there were no additional input, then this call would update s to $s' = x(sP)$ and return most of $x(s'Q)$. Given sQ, the attacker computes $sP = dsQ$, s', and $s'Q$. However, with additional input, the state s is updated to $s' = x((s \oplus \mathrm{H}(\mathsf{adin}))P)$ and then most of $x(s'Q)$ is returned. Given sQ, the attacker can compute $sP = dsQ$ and $x(sP)$ as before but there is no obvious way to obtain $(s \oplus \mathrm{H}(\mathsf{adin}))P$ from sP, even when adin is known.

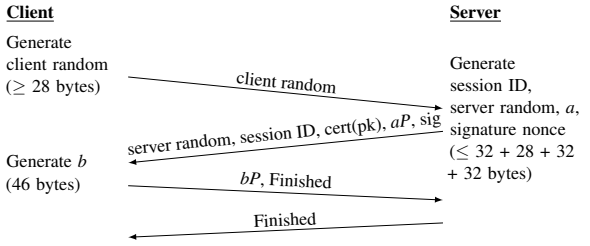
The addition of the final update step in Dual EC 2007 has the effect of making the basic attack possible when (1) the attacker has enough bytes from two consecutive calls; (2) the first call produces at most 30 bytes; and (3) additional input is used and guessable. In this case, the Dual EC 2007 state can be recovered while this is not possible for Dual EC 2006 due to the lack of an additional state update between the first and second call. In other cases, the extra state update means that attacking Dual EC 2007 is slightly slower than attacking Dual EC 2006.

Open questions. This theoretical analysis of Dual EC exploitability leaves open several obvious questions regarding the practical exploitability of Dual EC by network attackers who know the secret d. Do implementations of cryptographic protocols such as TLS actually expose enough Dual EC output to carry out the basic attack? How expensive are the computations required to compensate for missing output, and can these computations be made less expensive? Is additional input actually used, and if so is it hard to guess? Are certified implementations of Dual EC in fact implementing Dual EC 2006, Dual EC 2007, or something different?

The answers turn out to include several surprises, and in particular to rely on several implementation details and protocol details that have not been previously observed to be related to Dual EC. For example, the certified OpenSSL-FIPS implementation of Dual EC is actually non-functional, as mentioned earlier. As another example, although none of the attacks we implemented are in the case described above where exploitability relies on the difference between Dual EC 2006 and Dual EC 2007, the OpenSSL-fixed attacks are very close to this case: they avoid it solely because at one point OpenSSL-fixed calls Dual EC for 32 bytes made public through the TLS session ID, and 32 is larger than 30.



(a) TLS with RSA key transport.



(b) TLS with ECDHE exchange and ECDSA signature (P-256).

Figure 1: Simplified view of TLS handshakes.

Our analysis strongly suggests that, from an attacker's perspective, backdooring a PRNG should be combined not merely with influencing implementations to use the PRNG but also with influencing other details that secretly improve the exploitability of the PRNG. This paper does not attempt to determine whether this is what happened with Dual EC, and does not explore the difficult topic of defending against such attacks, beyond the obvious advice of not using Dual EC.

3 Attack target: TLS

TLS is the most widely used protocol for securing Internet communications [6]. TLS consists of several subprotocols, including a *record* protocol and *handshake* protocol. The handshake protocol is most relevant to the attacks discussed in this paper, so for simplicity we will describe only the relevant aspects of the handshake sub-protocol used in TLS version 1.2; further details are available in the RFC [6].

The handshake sub-protocol produces a fresh set of session keys with which application-layer data is encrypted and authenticated using the record protocol. Figure 1 de-

picts (simplified versions of) the two main handshakes for TLS: RSA key transport and ephemeral Diffie-Hellman key exchange (DHE). Ephemeral DHE uses either elliptic curve groups (ECDHE) or another suitable group. In either handshake, the client initiates a connection by sending a ClientHello message that includes a client nonce and a list of supported cipher suites. The server replies with a server nonce, a session ID, a certificate, an ephemeral DHE public key (for DHE), and a signature over the random nonces and public key. Signatures are either RSA or DSA. The specification mandates 32-byte client and server nonces, each consisting of 28 random bytes and a 4-byte timestamp. The construction of a session ID is arbitrary (i.e., up to the server implementation), though as we will see many implementations use the same PRNG that generates the ServerHello nonce and other cryptographic secrets. The client's next flow includes a client ephemeral DH public key (for DHE) or an RSA PKCS #1.5 encryption of a premaster secret (for RSA key transport). The premaster secret consists of either a 2-byte version number followed by a 46 byte random value (for RSA key transport), or the DHE secret defined by the DH public keys. Session keys are derived from the premaster secret and other values sent in the clear during the handshake, so learning the premaster secret is sufficient to break all subsequent encryption for a given session.

TLS extensions. There are many extensions to TLS, but we draw attention to four particular proposed — but not standardized — extensions. Each of these extensions has the side effect of removing the most obvious difficulty in exploiting Dual EC in TLS, namely the limited amount of randomness broadcast to the attacker. One might guess that these extensions make P-256 less expensive to exploit in TLS by a factor of 65,536 (and make P-384 and P-521 feasible to exploit), if they are actually implemented; our analysis in Section 4.1 shows that one of these extensions is in fact implemented in BSAFE, although the actual effect on exploitability is more complicated. None of these extensions have been previously described in connection with Dual EC.

One proposed extension, authored by Rescorla and Salter [25] in 2008, supports "Extended Random" client and server nonces. This extension is negotiable using the normal ClientHello extension mechanism, and includes up to $2^{16}-1$ bytes of data from a suitable PRNG. The server replies with its own Extended Random data that must be of the same length as the client's Extended Random data. The document states that this extension was requested by the United States Department of Defense with the claim that nonces "should be at least twice as long as the security level" (e.g., 256-bit nonces for 128-bit security).

Another proposed extension, "Opaque PRF" proposed by the same authors [24] in 2006, is nearly identical to Extended Random but does not require the data to be random. A third proposed extension, "Additional Random" by Hoffman [12] in 2010, is essentially the same as "Extended Random." After the initial announcement of our paper, Bodo Möller pointed out yet another similar extension, "Additional PRF Inputs" by Hoffman and Solinas [14] in 2009.

The "Internet-Drafts" describing these four extensions all expired without producing RFCs, although a generic framework [13] for such extensions was published as an Experimental RFC by Hoffman in 2012. IETF has not standardized any of these extensions.

Attack goals. We assume that the adversary's goal is to decrypt TLS packets to learn confidential material, or to steal long-lived secret keys. In the second case the secret keys need not be generated with Dual EC. We consider both small-scale *targeted* attacks and larger-scale *dragnet surveillance* attacks across broad swaths of the Internet.

Attack resources. Most of the attacks that we analyze are purely passive, relying solely on interception of TLS traffic sent through the network by the client and by the server. Usually seeing only one direction of TLS traffic is enough, and the attack can be mounted long after the fact using recorded connections. Occasionally an active attack is more powerful: for example, the range of μsecs in Section 4.3 becomes narrower if the attacker uses carefully timed connections (as in [29]) to more precisely pin down the server's clock.

The attacker is assumed to know the Dual EC back door d with $P = dQ$. All of the attacks rely on the client or server using Dual EC, but this is not an assumption; rather, it is something that we evaluate, by reverse-engineering several TLS implementations and also experimentally assessing the deployment of those implementations.

Our measurements of attack cost assume that the attacker knows the TLS software in use; otherwise the attacker has to try several of the attacks, increasing cost somewhat. See Section 6 for fingerprinting mechanisms.

The computer power required for attacking *one* Dual EC instance is very small by cryptanalytic standards: our optimized attacks (see Section 5) typically consume between $0.00001 and $1 of electricity, depending on the TLS implementation being attacked. (The exception to "typical" is OpenSSL; see Sections 4.3 and 5.2.) However, presumably the attacker's actual goal is to repeat the attack *many* times, especially in the dragnet-surveillance scenario. Our measurements allow straightforward extrapolations of the computer resources required for large-scale attacks.

4 Exploiting Dual EC in implementations

To attack each of the implementations discussed below, the attacker follows three basic steps: (1) recover Dual EC state from the session ID and/or server random fields in the TLS handshake; (2) compute the DHE or ECDHE shared secret which enables computing the 48-byte "master secret" from which all session keys are derived; and optionally (3) recover the long-lived DSA or ECDSA signing key used to sign the server's DHE or ECDHE public key.

Step (1) is an application of the basic attack which combines information exchanged in the handshake protocol messages to determine the correct Dual EC state from candidate states. Step (2) requires generating the DHE or ECDHE secret key by following the exact generation process used by the TLS implementation. Like Step (2), Step (3) duplicates the implementation's process for generating the nonce used in the signature of the public key. From the nonce, the signature, and the public key, it is straightforward to recover the signing key.

It is important to note that when a server uses DSA or ECDSA signatures, a *single* broken connection by a passive adversary is sufficient to recover the long-lived signing key which is used to authenticate the server's (EC)DHE public key. In contrast to RSA long-lived keys, recovering a server's (EC)DSA signing key does not enable future passive eavesdropping; it does allow impersonation of the server under active attack.

4.1 RSA BSAFE

Description. RSA's BSAFE family includes four libraries: Share for Java, Share for C and C++, Micro Edition Suite, and Crypto-J/SSL-J. We examined Share for Java and Share for C and C++. Although the two versions share a somewhat similar API, the implementation details differ, leading to different attacks.

The BSAFE family of libraries contains a number of options which can be configured at runtime. In order to avoid a combinatorial explosion in the number of configurations to test and attack, we focus our attention on the default configurations and the most secure cipher suites that lead to the use of the P-256 curve in Dual EC and, where applicable, ECDHE and ECDSA.[2]

Both BSAFE libraries we examined support both prediction resistance whereby the generator is reseeded on each call and output caching so that unused bytes from one call to generate can be used in subsequent calls rather than discarded. By default, neither option is enabled.

BSAFE-C. We examined the RSA BSAFE Share for C and C++ library (BSAFE-C) version 1.1 for Microsoft Windows. The library actually consists of two libraries: sharecrypto.lib implements the core cryptographic primitives, including Dual EC, and sharesslpki.lib implements TLS. Unlike the Micro Edition Suite, BSAFE-C is distributed only as static libraries with associated header files. This necessitated a minor reverse engineering effort to discover how BSAFE-C uses Dual EC in its TLS implementation.

Unlike the other TLS implementations we examined, BSAFE-C v. 1.1 does not support TLS 1.2. As a result, it does not support elliptic curve cryptography for either key exchange or digital signatures. By default, the preferred cipher suites are `TLS_DHE_DSS_WITH_AES_128_CBC_SHA` and `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` so we focused our efforts on these two.

A TLS server implemented using BSAFE-C generates several pseudorandom values used during the TLS handshake to establish session keys. In order, it generates (1) a 32-byte session identifier, (2) 28 bytes for the server random, (3) a 20-byte ephemeral Diffie–Hellman (DH) secret key, and, when using DSA, (4) a 20-byte nonce. The DH parameters and the server's public key are signed with the server's RSA or DSA certificate and the session ID, server random, public key, and signature are sent in the server's first flight of messages to the client during the handshake.

Although BSAFE-C's Dual EC interface does not cache unused output bytes by default, a separate, internal interface to produce pseudorandom values wraps Dual EC and provides its own layer of caching by only requesting multiples of 30 bytes from the Dual EC interface. This internal interface is used by all of the higher-level functionality, such as generating a DH secret key and a DSA nonce. Due to a quirk of the implementation, if a request to generate n bytes of output cannot be satisfied completely from the cached bytes, $\lfloor (n+29)/30 \rfloor \cdot 30$ bytes are generated in a single call to Dual EC, even if most of the n bytes will be taken from the cached bytes.

Caching output bytes means that when a new TLS session is started, an attacker who has not seen all prior connections has no way of knowing if the first value generated by the server — the session id — begins with a full output block or if it contains bytes cached from a previous call to Dual EC. However, due to the use of the requested number of bytes rather than the number of remaining bytes after pulling from the cache, the concatenation of the 32-byte session ID and the 28 pseudorandom bytes in the server random always contains a full 30-byte output block and between one and 30 bytes of a subsequent block where both blocks are generated in the call to Dual EC for 60 bytes made while generating the session ID.

A passive network attacker can easily recover the session keys and the server's long-lived DSA secret key used to sign the ephemeral DH parameters and public key. The attacker uses the publicly exchanged values in the

[2] Share for Java additionally supports P-384 and P-521 for Dual EC, ECDHE, and ECDSA.

connection through the ClientKeyExchange handshake message. At this point, the attacker knows the session ID, the client and server randoms, the DH parameters and client and server public keys, and the signature. This contains everything needed for the attack. The session keys are computed from the public values and the DH shared secret.

To recover the inner state of Dual EC a long string of consecutive output bytes is required. First, the session ID and the pseudorandom 28 bytes of the server random are concatenated into a 60-byte value B. Since up to 29 bytes of B can come from a previous call to Dual EC, one of $B[0..29], B[1..30], \ldots, B[29..58]$ must be a full output block. The basic attack is run on each in turn until the Dual EC state for the next output block is recovered. The attacker knows that the correct state has been found by (1) generating the next output block and comparing the corresponding bytes with the remaining bytes in B and then (2) generating more bytes as needed to produce a DH secret key and comparing the corresponding public key to the server's public key. If the public keys agree, then the DH shared secret can be computed hence the session keys can be computed.

Once the session keys for a single session have been recovered, more bytes can be generated to produce the DSA nonce. The DSA secret key a can be computed from the nonce k, public key (p,q,g,y), and the signature (R,S) of the message m as $a = R^{-1} \cdot \big(S \cdot k - H(m)\big) \bmod q$.

Recovering the Dual EC internal state requires performing approximately $30 \cdot 2^{15}$ scalar multiplications with a variable base point and an equal number with the fixed point Q, in the worst case. The total attack has a cost of $30 \cdot 2^{15}(C_v + C_f)$ where C_v (resp. C_f) is the cost of performing a single scalar multiplication with a variable (resp. fixed) base point. To generate the DH key, between $3C_f$ and $5C_f$ are needed to produce enough Dual EC output bytes; $1C_f$ is needed to compute server's DH public key; and, finally $1C_v$ is needed to compute the shared DH secret (using the client's DH input). Finally, to generate the nonce, at most one more Dual EC output is needed, using $3C_f$.

BSAFE-Java. We examined the RSA BSAFE Share for Java library version 1.1 (BSAFE-Java) and focused on connections using the `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` cipher suite.

Unlike BSAFE-C, the output from Dual EC is not cached so each generated output value is aligned with a block of generator output. Unfortunately (for the attacker), the session ID value produced by the server is not a 32-byte pseudorandom value. Instead, the attacker is forced to rely on the server random.

The values generated by Dual EC are, in order, (1) 28 bytes for server random; (2) 32 bytes for an ECDHE secret key; and (3) 32 bytes for an ECDSA nonce.

As before, a passive network attacker waits until she sees the ClientKeyExchange handshake message. At this point she has all of the information she needs to mount the following, simple attack. The 28 bytes from the server random are treated as bytes 2 through 29 of the 32-byte x-coordinate. She then mounts the basic attack by guessing the remaining most significant 16-bits and least significant 16-bits of the x-coordinate. A guess is checked by generating a 32-byte ECDH secret key, computing the corresponding public key, and comparing to the server's public key.

Once a match is found, the inner state of Dual EC is known. The session keys can be derived from the ECDH shared secret and the other values sent in the clear. Similarly, the ECDSA nonce can be found by generating another 32-byte value. As with the non-elliptic-curve DSA, the server's private key can be recovered from the nonce and the signature.

In the worst case, recovering the generator state requires approximately 2^{31} scalar multiplications with a variable base point and five times that number with a fixed base point to generate candidate ECDH secret keys and corresponding public keys. In total, the attack takes $2^{31}(C_v + 5C_f)$ work.

BSAFE connection watermarks and Extended Random. The documentation for BSAFE-C and BSAFE-Java indicate that they support connection watermarking and the TLS Extended Random extension described in Section 3.

In our experiments, BSAFE-Java has watermarks enabled by default. The watermark works by setting the first 20 bytes of the session ID to be the first 20 bytes of the server random and the last 12 bytes are set to the string "`RSA␣SSLJ␣␣␣␣`." This watermark can only be disabled by setting the property `com.rsa.ssl.server.watermark=disabled` in the Java security properties file [26].

We performed an Internet-wide scan of port 443 and found very few servers on this default port that exhibited this 32-byte watermark: only 386 of 8 million servers contacted. Details on this scan are included in section 6.

From reverse engineering the BSAFE-Java shareCrypto.jar library, we determined that it contained code to support or require the proposed TLS Extended Random extension; however, this code was disabled by means of a single static final boolean variable. We surmise that this code is not "dead" in the traditional sense, but rather the value of the variable can be changed to produce versions of the library with different features.

By changing the value of this variable, we were able to verify that the Extended Random extension is supported by the server.[3] When enabled and an Extended Random

[3] An analogous variable enables support for the client.

extension is received from the client, the server generates an equal length Extended Random response consisting of bytes generated by Dual EC concatenated with the same 12-byte watermark. The client Extended Random is 32-bytes by default. Interestingly, the 28 bytes for the server random and the Dual EC generated bytes for the Extended Random are generated together in a single call to Dual EC. As a consequence, any BSAFE-Java server which supports Extended Random exposes a sufficient quantity of contiguous output bytes to enable quick recovery of the session keys. There does not appear to be a mechanism for disabling the watermark in the Extended Random extension.

The BSAFE-C library documentation indicates that both watermarking and Extended Random are supported in some versions of the library; however, the version we have appears to have been compiled without this support.[4]

For both the Java and C versions of BSAFE, we have no evidence that versions of the libraries supporting Extended Random ever shipped. On the other hand, an earlier survey [1] found that Extended Random was occasionally requested by clients (about once in every 77000 connections). Our implemented attacks do not rely on Extended Random in any way.

4.2 Windows SChannel

SChannel ("Secure Channel") is a security component in the Windows operating system (introduced in Windows 2000) that provides authentication and confidentiality for socket-based communications. Although it supports several protocols, it is most commonly used for SSL/TLS, including by Microsoft's Internet Information Services (IIS) server and Internet Explorer (IE). We focus on ECDHE/ECDSA handshakes that use P-256 (which in turn cause Dual EC to also use this curve), as used by the version of IIS distributed with Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2. All information about the internal workings of SChannel and its implementation of Dual EC discussed in the following was obtained via reverse-engineering.

Description. SChannel uses Microsoft's FIPS 140-2 validated Cryptography Next Generation (CNG) API, which includes an implementation of Dual EC. CNG is implemented in two modules, one for user-mode callers (bcryptprimitives.dll) and one for kernel mode (cng.sys). Dual EC is used to generate pseudorandom bytes when the `BCryptGenRandom` function is explicitly directed to use it via a function argument or when it is selected as the system-wide default. When using Dual EC, `BCryptGenRandom` generates enough fresh blocks to satisfy the request, and discards any remaining bytes (i.e., there is no caching between requests).

Whenever SChannel requests random bytes, it calls `BCryptGenRandom` using the system-wide default. Our reverse-engineering efforts and experiments indicate that additional input is not provided by SChannel for TLS connections. TLS handshakes are performed by a separate process (lsass.exe) on behalf of IIS, which dispatches one of several worker threads to handle each request. Dual EC in CNG maintains separate state for each thread, so a successful attack on the state of one thread will not carry over to the others. Importantly, SChannel caches ephemeral keys for two hours (this timeout is hard-coded in the configurations we examined), and the cached keys are shared among all worker threads until the timeout expires.

When performing an ECDHE handshake, SChannel requests random bytes in a different order than OpenSSL and BSAFE (the number of bytes given in the following are specific to P-256 and some of them differ for other curves): (1) 32 bytes for session ID, (2) 40 bytes for ephemeral private key, (3) 32 bytes (not relevant to the attack), (4) 28 bytes for ServerHello nonce, and (5) 32 bytes for the signature (if using ECDSA). Notice the 40-byte request for the private key, even though a P-256 private key is only 32 bytes; this is because SChannel uses FIPS 186-3 B.4.1 (Key Pair Generation Using Extra Random Bits) to generate ECDHE key pairs, which specifies 8 additional bytes to reduce bias from a modulo operation. More importantly, SChannel requests bytes for the private key *before* the ServerHello random field. This means that any attempt to infer the private key must use the session ID, or random fields from previous handshakes.

Deviation from SP-800-90A. The implementation of Dual EC in CNG differs from the current SP-800-90A specification in one noteworthy way. The code in bcryptprimitives.dll that implements Dual EC (a function called `MSCryptDualEcGen`) seems to include the final update step at the end of each call — performing a point multiplication and projection on the x-coordinate after generating the necessary blocks. However, our reverse engineering efforts, as well as our experiments, indicate that the result is not copied into the seed state, and thus not used in subsequent calls to Dual EC. In short, although the CNG Dual EC implementation appears to contain code that implements the full current specification, it effectively implements Dual EC 2006 by ignoring the result of the final update step in future calls to generate. This appears to be a bug.

Fingerprint in the session ID. When an SChannel server generates a new session ID, it requests 32 bytes, $S[0,\ldots,31]$ from `BCryptGenRandom`, and interprets the first four bytes $S[0,\ldots,3]$ as an unsigned integer v. It then

[4] The header files for the version of BSAFE-C we have show that the library was compiled with the command line flags `-DNO_TLS_EXT_RAND -DNO_RSA_WATERMARK`.

computes $v' = v \bmod \texttt{CACHE\_LEN}$, and constructs the final session ID by concatenating these values, $session\_id = v'[0,\ldots,3] \| S[4,\ldots,31]$. CACHE_LEN is the maximum number of entries allowed in SChannel's session cache, which was hard-coded to 20,000 on the systems we tested. Thus, the presence of zeros in the third and fourth bytes of the session ID is a likely (although imperfect) fingerprint for SChannel implementations.

Attack 1: Using the server's random nonce. With Dual EC enabled, it is possible to use the 28-byte ServerHello nonce to learn the server's ECDHE private key, which will allow decryption of all ECDHE sessions within the two-hour window before the private key is refreshed. As previously discussed, these bytes are requested after the private key is generated, so in order to use them for the attack, we must look at previous handshake messages sent from the server. The fact that SChannel uses multiple threads to perform handshakes complicates the attack, as we cannot know which thread was used for a particular handshake unless we have learned the state of all threads and updated them as new handshakes were performed. On observing a handshake with the new server public ephemeral key, denoted h, the attacker works backwards through previous handshakes, using the random field in each ServerHello message to generate candidate Dual EC states using the basic attack. Each candidate state is checked first against the ECDSA public key to determine the state used in that handshake, and then against the session ID in h to determine if the same state was used to generate the new ephemeral key. The 32 bytes for the ECDSA nonce are generated in two calls, first 24 bytes then 8 bytes. These values are concatenated and then byte-wise reversed to obtain the nonce.

When the matching state is found, it is straightforward to generate the ephemeral private key and subsequent session keys. SChannel uses FIPS 186-3 B.4.1 to generate the private key, which corresponds to drawing 40 bytes of random input c, and computing the key as $(c \pmod{n-1})+1$, where n is the curve order. The worst-case complexity of this attack requires approximately 2^{31} scalar multiplications with a variable base point and four times as many with a fixed base point to check the ECDSA public key, totaling $2^{31}(C_v + 4C_f)$.

Attack 2: Using the session ID. The second approach uses the session ID in a handshake containing a new ECDHE public key. Denote the 32-byte session ID in the relevant handshake by S, and v' the unsigned integer corresponding to $S[0,\ldots,4]$. Recall that SChannel modifies the first four bytes of the session ID by replacing it with its value modulo CACHE_LEN. All that one must do to recover the private ephemeral key is run the basic attack on a set of inputs generated by enumerating (1) all 4-byte sequences whose unsigned integer representation v satisfies $v' = v \bmod \texttt{CACHE\_LEN}$ (for the first four bytes of the block that generated the session ID), and (2) all 2-byte sequences for the last two bytes of the first block that generated the session ID. Candidates are checked by generating the next 40 bytes, using FIPS 186-3 B.4.1 to construct a private key, and comparing the corresponding public key against that provided in the ServerKeyExchange.

This attack sidesteps the issues created by threading in SChannel, but because of the way the the session ID is generated it is actually more complex than the previous. Recall that $\texttt{CACHE\_LEN} = 20,000$ in both configurations tested, so this attack requires approximately 2^{18} guesses to deduce the first four bytes of the original session ID block, and 2^{16} for the last two bytes, giving approximately 2^{33} candidate curve points. Of these, approximately 2^{17} will agree with the last two bytes of the session ID, and we determine which is correct by generating two additional Dual EC blocks for a P-256 ECDHE private key, then performing a point multiplication to compare with the public key sent in the same handshake. The total complexity is $2^{33}(C_v + C_f) + 2^{17}(5C_f)$.

4.3 OpenSSL

Description. OpenSSL is one of the most widely used TLS libraries, due to its inclusion in many Linux/Apache distributions. While the standard edition of OpenSSL does not contain Dual EC, OpenSSL also ships a separate package called the OpenSSL FIPS Object Module. When this module is combined with OpenSSL, it provides a TLS library containing all four DRBG algorithms defined in NIST SP800-90A, including Dual EC. The Dual EC algorithm is not the default PRNG in OpenSSL, but it can be manually enabled by changing the PRNG settings through an API call at runtime.

Bug. While investigating the OpenSSL-FIPS implementation of Dual EC, we discovered a previously unknown bug that, in fact, prevented it from being run.[5] The presence of this bug may suggest that nobody has successfully run OpenSSL-FIPS configured to use Dual EC. However, the CMVP validation lists [22] show many "private" validations of the OpenSSL-FIPS module so it is possible that some commercial manufacturer has repaired this bug without propagating the fix back to the open source OpenSSL tree. For this reason, we felt it worthwhile to repair the bug in the FIPS module in order to investigate the feasibility of the attack.

Analysis of OpenSSL-fixed. We examined a repaired version of the OpenSSL FIPS Object Module ver-

[5] The bug involves a flaw in the runtime self-test mechanism that causes OpenSSL-FIPS to shut down the generator immediately upon initializing it. This bug is not triggered while the module is in TEST mode, which explains why unit and Known Answer Tests did not discover the issue. See [17] for details.

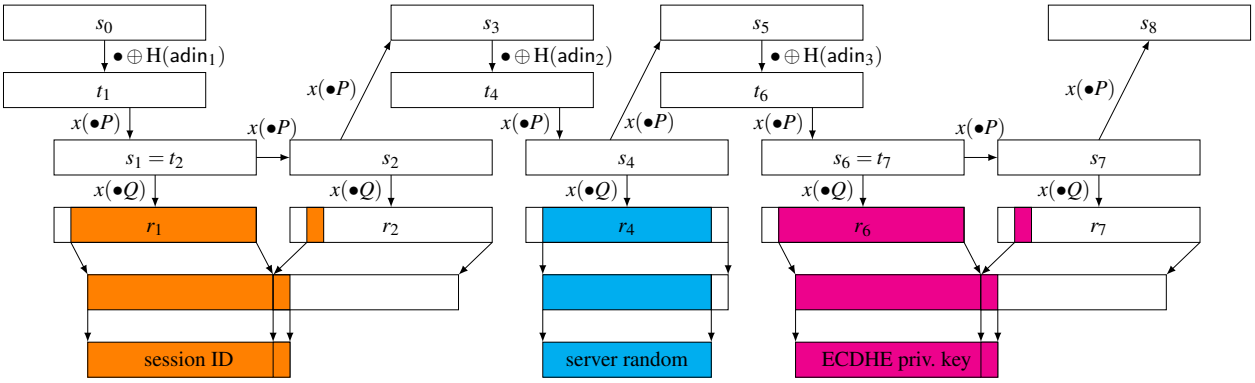


Figure 2: Dual EC usage in OpenSSL-FIPS using ECDHE with P-256.

sion 2.0.5 in combination with OpenSSL 1.0.1e (henceforth "OpenSSL-fixed"). The library consists of two components, libcrypto.a which implements the core cryptographic routines, including Dual EC, and libssl.a which implements TLS. OpenSSL documentation provides guidance on building the library, as well as usage in common scenarios.

OpenSSL-fixed supports TLS 1.2 with the full complement of elliptic curve cryptography for key exchange and digital signatures. By default, the preferred cipher suites use ECDHE key exchange and either RSA or ECDSA signatures. We investigated connections made using the ECDHE handshake.

OpenSSL includes a textbook implementation of Dual EC based on the NIST SP 800-90 March 2007 revision. On the server side of the standard ECDHE handshake, the generate function is called repeatedly to generate the following values: (1) a 32-byte session identifier, (2) a 28-byte server random,[6] (3) a 32-byte ECDHE ephemeral private key,[7] and, when ECDSA is being used, a 32-byte nonce. OpenSSL's implementation of Dual EC does not cache unused random bytes at the conclusion of a generator call, hence each sequence of random bytes begins with up to 30 bytes drawn from a single elliptic curve point. Figure 2 illustrates the generation of these values.

OpenSSL's use of additional input. While analyzing OpenSSL's implementation of SP 800-90, we discovered an important difference between OpenSSL and the other libraries analyzed in this work. Specifically, OpenSSL provides *additional input* with each call to the generate function. The additional input string is constructed uniquely by the function FIPS_get_timevec() prior to each query for random bytes. It comprises 16 bytes with the following structure.

$$\mathsf{adin} = (\text{time in secs} \parallel \text{time in } \mu\text{secs} \parallel \text{counter} \parallel \text{pid})$$

Each of the component fields in the additional input string is 4 bytes in length. On Unix-based systems the time fields are computed using gettimeofday(). The counter is a monotonically increasing global counter that is set to 0 at library initialization, and increments with each call to FIPS_get_timevec(). On operating systems where the process IDs are available, pid contains the process ID returned from getpid().

A passive attacker can capture 32 consecutive bytes of Dual EC output by observing the session ID sent to the client by an OpenSSL server. Assuming the generator is instantiated with P-256, the attacker can now execute the initial steps of the basic attack using the first 30 bytes, in order to recover multiple candidate states, and (using the additional two bytes) reduce the number of candidate states to one, or a small number. From this point, the OpenSSL attack differs from the basic attack. Given each candidate state s, the attacker now calculates the final update step $s = x(sP)$ and exhaustively guesses the additional input string used in the next call to the generate function as $s' = s \oplus \mathrm{H}(\mathsf{adin})$. This requires the attacker to iterate through a set of candidate adin input strings, executing the steps of the generate algorithm to recover a candidate ECDHE private key, and comparing this value to the intercepted ECDHE public key from a real handshake trace.

The complexity of this attack depends on two factors: the number of candidate states remaining at the conclusion of the first portion of the attack, and the number of candidate adin strings. Since we are guessing 16-bits, only about half of all strings give a valid x-coordinate, and are comparing the resultant output against 16 bits, we expect to see 1 or 2 candidate states that generate the correct first two values. We did not see more than 3 candidate states in any of our test runs, although we would

[6] Although we do not discuss attacks against the client, a recent fix to the OpenSSL client implementation increases the amount of PRNG output in the client random to 32 bytes (see `http://bit.ly/1ftSQrM`) which may decrease the attack complexity significantly.

[7] OpenSSL generates this key by drawing 32 random bytes and checking whether the result (expressed as an integer) is less than the group order n. If not, the process is repeated.

expect to occasionally see more if we repeated the attack enough times.

Since the time in seconds is already transmitted as part of the server random, the first portion of adin is known. Thus it remains to predict the time in μseconds, process ID and counter. Under reasonable assumptions about the operating system and the number of connections so far handled by the server, this can range from approximately 2^{20} (primarily guessing the μsecs field) to 2^{35} with a typical Unix range of pid values and known counter value, and possibly 2^{45} or more depending on how recently the library was initialized. Notice that once an attacker recovers the adin string for a first TLS connection, it may be relatively easy to predict these values for later connections.

The inclusion of additional input complicates the attack since recovering the Dual EC state when it is most convenient, namely during the generation of the session ID, does not immediately translate into recovering the session keys. There are two cases to consider.

In the first case, the attacker knows nothing about the state of the generator except that the counter value is no bigger than $k \leq 32$ bits. The first step is to recover the generator state (for ease of analysis, assume only one candidate state is possible). As with BSAFE-C, this requires approximately 2^{15} variable-base-point multiplications and an equal number of fixed-base-point multiplications. Next, the additional input string needs to be guessed. For each guess, this takes two fixed-base-point multiplications. There are at most 2^{35+k} additional input strings to try. A guess can be validated by comparing to the server random field. Finally, the ECDHE secret and public keys need to be computed for each guess of the second additional input string. Each guess takes five fixed-base-point multiplications; however, since the attacker has already determined the pid and the counter value, the attacker has a good estimate of the time and increments the microsecond value from there; this takes about 2^{13} guesses. This gives a total cost of $2^{15}(C_v + C_f) + 2^{35+k}(2C_f) + 2^{13}(5C_f)$. The 2^{13} is an upper bound for our observations. Usually fewer than 2^{12} tests were sufficient and on a fast Internet server even less time passes between two calls of Dual EC.

In the second case, the attacker has already broken a previous connection and so the pid and counter values are known. The cost of performing the whole attack a second time becomes $2^{15}(C_v + C_f) + 2^{20}(2C_f) + 2^{13}(5C_f)$. However, the cost of computing a scalar multiplication with a variable base point is significantly higher than for a fixed base point. It may be in the attacker's best interest to keep track of the generator's state throughout each session. This involves keeping track of counter updates and recovering the state after each encrypted TLS record sent and randomness used for ECDSA and IVs. The search space for the time in adin for these values is usually small, similar to that in the ECDHE key.

Then the cost of recovering the state at the beginning of a new connection is at most $2^{20}(2C_f)$ for testing the time (and less if better estimates of the time are known) in place of the $2^{15}(C_v + C_f)$, for a total cost of $2^{20}(2C_f) + 2^{13}(7C_f)$. This is faster if the time update for the server random call requires a smaller search space for the time after the time has been determined for the session ID.

4.4 Attack validation

We implemented each of the attacks against TLS libraries described above to validate that they work as described. Since we do not know the relationship between the NIST-specified points P and Q, we generated our own point Q' by first generating a random value $e \xleftarrow{R} \{0, 1, \ldots, n-1\}$ where n is the order of P, and set $Q' = eP$. This gives our trapdoor value $d \equiv e^{-1} \pmod{n}$ such that $dQ' = P$. We then modified each of the libraries to use our point Q' and captured network traces using the libraries. We ran our attacks against these traces to simulate a passive network attacker.

We would like to stress that anybody who knows the back door for the NIST-specified points can run the same attack on the fielded BSAFE and SChannel implementations without reverse engineering.

We describe the concrete performance results of our attacks in the next section and give details on the libraries here.

RSA BSAFE. The Dual EC implementation in BSAFE-C contains the points P and Q as well as three tables of scalar multiples of each of the points for fast multiplication. The tables contain 65, 517, and 573 multiples. After working out the corresponding scalar factor for each entry in the tables, we computed our own tables and modified the relevant object files in the library. There were no health checks or known-answer tests (KATs) to bypass.

BSAFE-Java is distributed as a signed, obfuscated jar file. We reverse engineered the code sufficiently to find and bypass the checks that prevent modification and replaced the jar's signature with our own. BSAFE-Java has a single table of 431 scalar multiples of each of P and Q.

Windows SChannel. Dual EC in SChannel is implemented both in the kernel and a user-mode library. We modified the user-mode library, which performs a KAT when the operating system first loads the module at boot, as well as continuously during operation when FIPS mode is enabled. To sidestep these checks, we disabled FIPS mode, and wrote a system service that (1) replaces Q with Q' in the the address space of the Local Security Authority Subsystem Service (IIS and IE delegate TLS handshakes to this process), and (2) makes Dual EC the system-wide default PRNG.

OpenSSL-fixed. Dual EC in OpenSSL is implemented in the separate OpenSSL-FIPS library. This library contains both runtime KATs and a check of the SHA-1 hash of the object code. Since the hash is computed each time the library is compiled, we simply fixed the bug which prevents Dual EC from being used (described above), bypassed the KATs, and substituted Q' for Q.

5 Implementation

We implemented all attacks for parallel architectures, specifically clusters of multicore CPUs. The attacks are parallelized using OpenMP and MPI, with the search space distributed over all cores of the cluster nodes, using one process per CPU and one thread per (virtual) core. The attacks are "embarrassingly parallel": there are no data dependencies between the parallel computations and thus no communication overhead and no limit on the scalability of the parallelization, other than the total number of independent computations.

5.1 Algorithmic optimizations

For finite-field arithmetic, we use the Gueron/Krasnov OpenSSL patch for NIST P-256, described in [10] and available at [11]. Square-root computations, to recover the y-coordinates, use that $p \equiv 3 \bmod 4$ and compute $\sqrt{a}$ as $a^{(p+1)/4}$. We refer to the cost of recovering a y-coordinate as C_y.

The definition of the update function in Dual EC requires all scalar multiplications to result in affine points (to derive a unique x-coordinate). To improve the performance of our implementation we compute all point operations in affine coordinates and batch the inversions using Montgomery's trick [20] across several parallel computations. We use a batch size of 256 for all experiments; increasing the batch size any further does not have a measurable effect on the runtime.

The most performance-critical operations on EC points in the attack logic are:

1. Scalar multiplications using fixed base points P and Q in order to compute the next internal state and to compute the output string respectively; P is also used as base point for ECDHE and ECDSA computations.

2. Scalar multiplications using variable base points and a fixed scalar, the back door d, in order to compute a candidate internal state given an output string.

In Table 1 we refer to the costs of a fixed-base-point scalar multiplication as C_f and those of a variable-base-point one as C_v.

For the fixed-base-point computations we use large precomputed tables of multiples of the base point. For a given width w we compute a lookup table consisting of $T_{P,i,j} = i2^{jw}P$ for $0 < i < 2^w$, $0 \le j < \lceil 256/w \rceil$. A scalar multiplication sP can then be performed as $\lceil 256/w \rceil - 1$ additions of precomputed points from the lookup table using $sP = \sum_{j=0}^{\lceil 256/w \rceil - 1} T_{P,s(j),j}$, where $s = \sum_{j=0}^{\lceil 256/w \rceil - 1} s(j)2^{jw}$. We do the same for Q in place of P. These tables are shared among all threads of each process in the implementation. We choose $w = 16$ for all our experiments for a reasonable balance between performance and lookup-table size. This brings C_f down to 15 point additions.

We implemented the scalar multiplications with the fixed scalar d using signed sliding windows with window width 5 and fully unrolled the code. This way C_v takes 253 doublings and 50 additions. Our d was a randomly chosen 256-bit integer. An attacker can choose d to minimize the cost of the fixed-scalar variable-base-point scalar multiplication by choosing d with low Hamming weight or more generally with a short addition chain, although a sufficiently low weight runs the risk that someone will discover d by a discrete-logarithm computation. To put an upper bound on the Dual EC attack time we avoid this optimization.

An independent blog post by Adamantiadis [2] has a proof of concept of the general Dual EC attack using OpenSSL's libcrypto for curve and large integer arithmetic. Adamantiadis does not implement a complete attack but recovers the state from a 30 byte random output. His proof of concept iterates through all 2^{16} candidates to recover the missing bits of the x-coordinate and computes the corresponding y coordinate. In case he discovers a point on the curve, he applies the back-door computation and computes the next random output. This proof of concept has an expected cost of $2^{16}C_y + 2^{15}(C_v + C_f)$.

On a single core of an Intel Xeon CPU E3-1275 v3, Adamantiadis's code requires about 18.5 s compiled with gcc version 4.8.1. Adamantiadis is using an older version of OpenSSL's libcrypto. For comparison, we modified Adamantiadis's code to run with libcrypto from OpenSSL version 1.0.1e; this version requires about 12.1 s. Furthermore, we reimplemented Adamantiadis's proof of concept using our optimized primitives. The optimized version requires about 3.7 s on a single core. Thus, our optimizations give an improvement by a factor of 3.3 over libcrypto.

In Adamantiadis's code (using libcrypto version 1.0.1e), the computation of a y coordinate (corresponding to cost C_y) takes about 15 µs on average. In our optimized version, this computation requires only 6 µs, which is an improvement by a factor of 2.5. The application of the back-door computation in Adamantiadis's code (scalar multiplication of a variable point by a fixed factor, cost C_v) requires about 168 µs on average; our code requires about 98 µs, which is an improvement by a factor of 1.7. Scalar multiplication with fixed base points P and Q (cost C_f) benefits the most from our optimizations. In

Table 2: Performance measurements and estimates.

Attack	Intel Xeon Reference System			16-CPU AMD Cluster
	2^{22} Candidates (s)	Expected Runtime (min)	Expected Cost	Total Runtime (min)
BSAFE-C v1.1	–	0.26	16	0.04*
BSAFE-Java v1.1	75.08*	641	38,500	63.96*
SChannel I	72.58*	619	37,100	62.97*
SChannel II	62.79*	1,760	106,000	182.64*
OpenSSL-fixed I	–	0.04	3	0.02*
OpenSSL-fixed II	–	707	44,200	83.32*
OpenSSL-fixed III	–	$2^k \cdot 707$	$2^k \cdot 44{,}200$	$2^k \cdot 83.32$

*measured

Adamantiadis's code, one scalar multiplication requires about 171 µs on average. In our optimized code, the computation for fixed base points requires only about 6µs on average, which is an improvement by a factor of 28. For an actual attack, the proportion of C_f to C_v is usually significantly larger than in the proof of concept. This increases the impact of our improvements on the attacks.

5.2 Performance measurements and estimates

All our attacks are based on the fact that some fields in the handshake messages (e.g., session ID and server random) contain a bit sequence derived from the x-coordinate of a point R. In order to recover R, we iterate through all possible combinations of the missing bits, check whether each candidate r_i actually is a valid x-coordinate and gives a point candidate R_i, apply the back door by computing dR_i, and follow all the steps (including adin for the attacks on OpenSSL-fixed) to check whether the candidate r_i eventually allows us to recover the (EC)DH secret. As the steps differ for each implementation, a different amount of computation is required for each attack (see Table 1, column "Attack Complexity").

We measure the cost of the attacks on a reference CPU, an Intel Xeon CPU E3-1275 v3, which has 4 cores and 2 hardware threads per core when enabling Hyper Threading. Table 2 lists measured and estimated performance numbers of the attacks. Turbo Boost and Hyper Threading are enabled; thus, we were using 8 OpenMP threads for the measurements on the reference system.

We measure the runtime of testing 2^{22} candidates (about 2^{21} candidate points). From these measurements, we extrapolate the expected runtime of the attack. From the expected runtime, we compute the cost of the attack as the number of Intel Xeon reference processors that would be required to perform the attack in an expected time of less than one second.

Finally, to verify the efficiency of the attack on multiple nodes, we measure the total worst-case runtime of the attack on a four-node, quad-socket AMD Opteron 6276 (Bulldozer) computing cluster. The cluster has an Infiniband interconnect and 256 GB memory per node — however, neither of these is relevant for the attacks: the attacks require less than 1 GB of RAM per process and do not need much communication.

For the timing measurements we ran each case several times to verify that there is no significant variance and finally picked the time from a representative test run. We are using gcc version 4.8.1 (Ubuntu/Linaro) with optimization level O3. In the following, all estimates for expected runtime and expected cost are rounded to three significant digits.

BSAFE-C v1.1. For the BSAFE-C attack, we simply concatenate session ID and server random and guess 16 bits of the target x-coordinate for the 30 possible cases. The complexity of the attack on BSAFE-C is $30 \cdot \left(2^{16} C_y + 2^{15}(C_v + C_f)\right)$. In the worst case this only requires testing $30 \cdot 2^{16}$ candidates which is less than 2^{22}, so we do not have a measurement for the first column in Table 2. Instead, we measured the worst-case time for the whole attack (31.12 seconds) and list half of the worst-case time, i.e., $31.12\ \mathrm{s}/2 \approx 0.26$ min as expected runtime. This gives an expected cost of 16 reference CPUs. This attack required 0.04 min on our cluster; most of this time is probably due to initialization overhead.

BSAFE-Java v1.1. In this case, the session ID of the handshake is not derived from Dual EC — so we have to use the 28 bytes of the server random, missing 32 bits of the target x-coordinate. The complexity of the attack on BSAFE-Java is $2^{32} C_y + 2^{31}(C_v + 5C_f)$. We measured a time of 75.08 s to check 2^{22} candidates. In total, this attack requires checking at most 2^{32} candidates, so the expected runtime is $2^{32-22} \cdot 75.08\ \mathrm{s}/2 \approx 641$ min on the reference CPU. Therefore, the expected cost to finish this attack within one second is about 38,500 reference CPUs.

We measured a worst-case total runtime of 63.96 min on our cluster.

SChannel I. The SChannel I attack uses the server random from the preceding handshake to hook into the random number stream and to discover the server's ECDHE private key in the handshake when the private key is refreshed. The complexity of this attack is $2^{32}C_y + 2^{31}(C_v + 4C_f)$. Checking 2^{22} candidates takes 72.58 s. This is slightly less than the time for BSAFE-Java, because this attack requires only four instead of five multiplications by a fixed base point for each point candidate. The whole attack requires checking at most 2^{32} candidates, so the expected runtime is $2^{32-22} \cdot 72.58\ \text{s}/2 \approx 619$ min. Therefore, the expected cost of the attack is 37,100 reference CPUs. The measured worst-case total time on our cluster is 62.97 min.

SChannel II. The SChannel II attack uses just one single handshake to recover the secret keys and therefore relies on the session ID (where the 4 least significant bytes have been replaced by their value modulo 20,000) to recover the state of the PRNG. The complexity of the attack is $2^{34}C_y + 2^{33}(C_v + C_f) + 2^{17}(5C_f)$, more precisely $2^{32}/20{,}000 \cdot 2^{16}\left(C_y + (C_v + C_f + 5C_f/2^{16})/2\right)$. The dominant part for each candidate check is $C_y + (C_v + C_f)/2$ which requires a smaller number of multiplications with a fixed base point than SChannel I. We measured 62.79 s to check 2^{22} candidates. This attack requires checking up to $2^{32}/20{,}000 \cdot 2^{16}$ candidates; therefore, this attack has an expected runtime of $2^{32-22}/20{,}000 \cdot 2^{16} \cdot 62.79\ \text{s}/2 \approx 1{,}760$ min. This gives an expected cost of 106,000 reference CPUs.

OpenSSL-fixed. Due to the use of adin before each random draw, OpenSSL is a special case among the implementations of Dual EC. The attack on OpenSSL takes three steps: First, we find the current state by finding the 16 missing bits for the session ID. This requires checking at most 2^{16} candidates; thus, we do not give a measurement for 2^{22} candidates in the first column of Table 2. Since this step might result in more than one state candidate, we always compute all 2^{16} candidates. If more than one candidate is recovered, the attacker either has to check all candidates (in parallel) or retry with a different handshake if applicable. In the following we investigate the expected case that only one candidate is found. In the second step, we need to find the adin used to generate server random. Here, adin consists of the current system time (including µs), the process ID (pid), and a counter value. In the last step, we need to find the next adin before the call to generate the DH key. The pid and the counter are known from the previous adin; we only need to find the µs over a very short time span by iteratively incrementing the time counter until the correct value is reached. The complexity of the OpenSSL attack is $2^{16}C_y + 2^{15}(C_v + C_f) + 2^{20+k+l}(2C_f) + 2^{13}(5C_f)$ where k is the number of unknown bits of the adin counter and l is the number of unknown bits of the adin pid.

The first step requires to check at most 2^{16} x-coordinate candidates; for the last step, we expect a maximum of 2^{13} increments to find the correct µs for the adin, as discussed earlier. Therefore, the scalability of the parallelization of step one and three is limited due to the small workload. We are using a batch size of 256. Therefore, the workload of checking 2^{16} candidates in the first step can be split across at most $2^{16}/256 = 256$ threads without loss of efficiency. The last step requires at most 2^{13} iterations, so the maximum number of threads for this step is $2^{13}/256 = 32$. Step one and three contribute to only an insignificant fraction of the total complexity when pid or counter are not known.

We examine three cases:

OpenSSL-fixed I: pid and counter are known, µs of time are unknown,

OpenSSL-fixed II: counter is known, µs of time and pid (15 bits) are unknown,

OpenSSL-fixed III: µs of time, pid (15 bits), and counter (k bits) are unknown.

The system time in seconds is known from the timestamp in the server-random field of the handshake message. Therefore, only the µs must be found by exhaustive search. The seconds might have clocked since the timestamp was obtained; thus, we need to test up to $1{,}000{,}000 + \Delta$ candidates for the µs. An upper limit on Δ is the time between the server receiving the ClientHello message and sending the ServerHello message. We use $1{,}000{,}000\ \mu\text{s} + 48{,}576\ \mu\text{s} = 2^{20}\ \mu\text{s}$ as upper limit.

The standard maximum pid on Linux systems is 2^{15}. If the attacker starts listening to the server right after bootup, he can assume the initial counter to be zero; otherwise, he may make an educated guess about the current counter state based on uptime and the average connection number.

To compute the expected runtime of this attack we measured the worst-case runtime of the case OpenSSL-fixed I. The first step to compute state candidates took about 0.96 s; the second step checking all possible 2^{20} µs for one single state candidate took 2.59 s. The final step checking the next 2^{13} µs took only 0.05 s. Therefore, the expected runtime of OpenSSL-fixed I is $0.96\ \text{s} + 2.59\ \text{s}/2 + 0.05\ \text{s}/2 \approx 0.04$ min; the expected cost is three reference CPUs. The expected runtime of OpenSSL-fixed II is $0.96\ \text{s} + 2^{15} \cdot 2.59\ \text{s}/2 + 0.05\ \text{s}/2 \approx 707$ min. We are using 8 threads in the reference system; the maximum number of threads is 256 threads for the first step and 32 threads for the last step. Therefore, the first step cannot be faster than $0.96\ \text{s}/(256/8) \approx 0.03$ s and the last step requires at least $0.05\ \text{s}/(32/8) \approx 0.01$ s in the worst

case. To finish the whole attack in one second on average, the second step must take $1\text{ s} - 0.03\text{ s} - 0.01\text{ s} = 0.96\text{ s}$ on average which requires $2^{15} \cdot 2.59\text{ s}/2/0.96\text{ s} \approx 44{,}200$ reference CPUs. For OpenSSL-fixed III these values are multiplied by 2^k, assuming k unknown bits for the counter of the adin.

We ran OpenSSL-fixed II on the cluster, testing all 2^{35} combinations of µs and pid to obtain the worst-case total runtime. The time of 83.32 min is noticeably less than 2^{15} times the time of 0.02 s taken in the OpenSSL-fixed I scenario. This is because in OpenSSL-fixed I the computations of $2^{15}C_v$ for the first step have a strong impact on the runtime; testing 2^{35} candidates gives more precise timing estimates of C_f. We can extrapolate the costs for OpenSSL-fixed III as 2^k times those of OpenSSL-fixed II because the contribution of the first and of the third step become negligible.

Summary. These runtime and cost estimates show that a powerful attacker (in case of BSAFE-C, an arbitrary attacker) is able to break TLS connections that use the Dual EC pseudorandom number generator when he possesses the back-door information. The usability of the attack on OpenSSL-fixed depends on additional knowledge about the adin; however, computing clusters of around 100,000 CPUs are realistic as of today (for example the Tianhe-2 supercomputer in China has 16,000 computing nodes with 5 CPUs each [19]) and sufficient to break BSAFE and SChannel in less than one second.

6 Passive TLS server fingerprinting

In many contexts, including exploitation of the Dual EC backdoor, it is useful to identify, or fingerprint, the implementation used by a TLS server. Existing tools for TLS fingerprinting use active techniques (requesting a page to get an error message and analyzing the result), but our investigations of TLS implementations suggest that the session ID field, in particular, admits a passive fingerprinting mechanism useful to an attacker observing network traffic or even one attacking recorded connections from years ago.

Data collection. We collected a large dataset of TLS session information from servers listening on port 443 in the IPv4 address space. We executed a ZMap scan [7] of port 443 over the entire IPv4 address space (excluding ZMap's default blacklist). The ZMap scan netted 38.9 million services responding on port 443. For 37.1 million of these services, we used a modified version of OpenSSL v1.0.1e s_client to connect to the service, and attempt to perform a TLS handshake up through receiving the ServerHello message (containing the session ID, server random value, and TLS server extensions), and then sent a TCP RST to the server. Of these attempts, 21.8M servers responded with a ServerHello message.

We investigated a number of candidate fingerprints based on observable behavior to a passive adversary. For each server that exhibited the RSA BSAFE fingerprint, we made an HTTP GET request on port 443 in an attempt to determine what software the server uses via the self-reported `Server` field of the HTTP header. We repeated this for 1,000 randomly selected IP addresses exhibiting the SChannel fingerprint. We consider an observable behavior to be a *selective fingerprint* if $\geq 95\%$ of the servers from which we received HTTP headers identify themselves as the same implementation.

6.1 Fingerprints detected

We detected many different types of fingerprints by examining server random values, session IDs, and TLS server extensions (all unencrypted values to a passive observer). In addition to the fingerprints on BSAFE and SChannel discussed in Sections 4.1 and 4.2, we identified five selective fingerprints from unique combinations of supported extensions, 2 selective fingerprints corresponding to session ID values with fewer than 32 bytes, and seven selective fingerprints corresponding to fixed subsequences in the session ID.

In sum, 4 million of the servers we contacted exhibited selective fingerprints. We discuss our findings for BSAFE and SChannel in more detail below.

RSA BSAFE. As described in Section 4.1, by default, BSAFE-Java has a very prominent fingerprint that is enabled by default, and BSAFE-C has a similar fingerprint that is not enabled by default. We found 720 servers with the BSAFE-Java fingerprint, and none with the BSAFE-C fingerprint. Of these servers, 33% self-reported running Apache Coyote 1.1,[8] with the remaining two self-reported implementations ("ADP API" and lighthttpd) appearing on fewer than ten instances. The remaining servers did not return a `Server` field.

Microsoft SChannel. As described in Section 4.2, SChannel exhibits a fingerprint in the first 4 bytes of the session ID. 2.7 million of the servers we contacted exhibited this fingerprint. We requested HTTP headers from 1,000 of these IPs (randomly selected), and 96% of the responses included the string "Microsoft" in the server field, suggesting that this is a selective fingerprint.

7 Conclusions

We provided the first theoretical and practical analysis of the exploitability of Dual EC as used in deployed TLS implementations. We evaluated the viability and performance of recovering TLS session keys for fielded implementations that use Dual EC. Our results demonstrate that

[8] Apache Coyote is a front end that forwards requests to Apache Tomcat, which supports Java Servlets and JavaServer pages; running Tomcat with BSAFE-Java may indicate an effort to provide a FIPS-compliant web application.

otherwise innocuous implementation decisions greatly affect exploitability. For example, RSA BSAFE-C is by far the easiest to exploit due to caching of unused bytes of Dual EC output. On the other end of the spectrum, OpenSSL-fixed uses additional input, which can render attacks significantly more challenging if no or only little information is available about the server.

We developed and successfully tested state-of-the-art parallelized implementations of all attacks against versions of the libraries patched to use Dual EC constants that we generated. Depending on the design choices in the implementations, an attacker can recover TLS session keys within seconds on a single CPU or may require a cluster of more than 100,000 CPUs for the same task if a different library is used. For OpenSSL some parameters might require such a serious cluster for an even longer time.

While there are a number of available mitigations to the vulnerabilities we discuss in this work, the simplest and best is to remove the Dual EC implementation from deployed products. OpenSSL has already initiated the (expensive, due to FIPS certification) process of removing Dual EC from its FIPS version and, in the meantime, is not fixing the bug we discovered that prevents its use [17]. RSA has advised developers to stop using the BSAFE Dual EC implementation [9]. Our work further emphasizes the need to deprecate the algorithm as soon as possible.

Acknowledgements

We would like to thank Bo-Yin Yang's research group at Academia Sinica, Taiwan, for giving us access to their computing cluster for benchmarking. We also thank Steve Marquess and Dr. Stephen Henson of the OpenSSL Foundation for helpful discussions. We thank Kurt Opsahl and Nate Cardozo of the Electronic Frontier Foundation for their legal advice throughout this project. Adam Everspaugh contributed to Section 6 of this paper.

This work was supported in part by the European Commission through the ICT program under contract INFSO-ICT-284833 (PUFFIN), by the US National Science Foundation under grants 1018836, 1314919, and 1065134, by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, and by a gift from Microsoft.

References

[1] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Revisiting SSL: A large-scale study of the Internet's most trusted protocol, 2012. Online: http://www1.icsi.berkeley.edu/~bernhard/papers/ICSI_TR-12-015.pdf.

[2] Aris Adamantiadis. Dual_EC_DRBG backdoor: a proof of concept, December 2013. Online: http://blog.0xbadc0de.be/archives/155.

[3] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how US and UK spy agencies defeat internet privacy and security. *The Guardian*, September 5 2013. Online: http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security.

[4] Daniel R. L. Brown and Scott A. Vanstone. Elliptic curve random number generation, August 2007. Online: http://www.freshpatents.com/Elliptic-curve-random-number-generation-dt20070816ptan20070189527.php.

[5] Art Coviello. RSA Conference 2014 keynote for Art Coviello, February 2014. Online: http://www.emc.com/collateral/corporation/rsa-conference-keynote-art-coviello-feburary-24-2014.pdf.

[6] Tim Dierks and Eric Rescorla. RFC 5246: The transport layer security (TLS) protocol version 1.2. The Internet Engineering Task Force, August 2008. Online: http://tools.ietf.org/html/rfc5246.

[7] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings USENIX Security 2013*, pages 605–619, 2013.

[8] Kristian Gjøsteen. Comments on Dual-EC-DRBG/NIST SP 800-90, draft December 2005, 2006. Online: http://www.math.ntnu.no/~kristiag/drafts/dual-ec-drbg-comments.pdf.

[9] Dan Goodin. Stop using NSA-influenced code in our products, RSA tells customers, September 2013. Online: http://arstechnica.com/security/2013/09/stop-using-nsa-influence-code-in-our-product-rsa-tells-customers/.

[10] Shay Gueron and Vlad Krasnov. Fast prime field elliptic curve cryptography with 256 bit primes. *IACR Cryptology ePrint Archive*, 2013:816, December 2013. Online: http://eprint.iacr.org/2013/816.

[11] Shay Gueron and Vlad Krasnov. Fast and side channel protected implementation of the NIST P-256 elliptic curve, for x86-64 platforms, 2013. Online: https://rt.openssl.org/Ticket/Display.html?id=3149&user=guest&pass=guest.

[12] Paul Hoffman. Additional random extension to TLS, February 2010. Online:

http://tools.ietf.org/html/draft-hoffman-tls-additional-random-ext-01. Internet-Draft version 01.

[13] Paul Hoffman. RFC 6358: Additional master secret inputs for TLS. The Internet Engineering Task Force, January 2012. Online: http://tools.ietf.org/html/rfc6358.

[14] Paul Hoffman and Jerome Solinas. Additional PRF inputs for TLS, October 2009. Online: http://tools.ietf.org/html/draft-solinas-tls-additional-prf-input-01. Internet-Draft version 01.

[15] Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 27, IT Security techniques. ISO/IEC 18031: Information technology – Security techniques – Random bit generation, 2005.

[16] Jeff Larson, Nicole Perlroth, and Scott Shane. Revealed: The NSA's secret campaign to crack, undermine Internet security. *ProPublica*, September 2013. Online: http://www.propublica.org/article/the-nsas-secret-campaign-to-crack-undermine-internet-encryption.

[17] Steve Marquess. Flaw in Dual EC DRBG (no, not that one), December 2013. Online: http://marc.info/?l=openssl-announce&m=138747119822324&w=2.

[18] Joseph Menn. Exclusive: Secret contract tied NSA and security industry pioneer. *Reuters*, December 2013. Online: http://www.reuters.com/article/2013/12/20/us-usa-security-rsa-idUSBRE9BJ1C220131220.

[19] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. TOP500 supercomputer sites, June 2013. Online: http://www.top500.org/lists/2013/06/.

[20] Peter Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48: 243–264, 1987.

[21] National Institute of Standards and Technology. Special Publication 800-90: Recommendation for random number generation using deterministic random bit generators, 2012. Online: http://csrc.nist.gov/publications/PubsSPs.html#800-90A. (first version June 2006, second version March 2007).

[22] National Institute of Standards and Technology. DRBG validation list. February 2014. Online: http://csrc.nist.gov/groups/STM/cavp/documents/drbg/drbgval.html.

[23] Nicole Perlroth, Jeff Larson, and Scott Shane. N.S.A. able to foil basic safeguards of privacy on web. *International New York Times*, September 2013. Online: http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html.

[24] Eric Rescorla and Margaret Salter. Opaque PRF inputs for TLS, December 2006. Online: http://tools.ietf.org/html/draft-rescorla-tls-opaque-prf-input-00. Internet-Draft version 00.

[25] Eric Rescorla and Margaret Salter. Extended random values for TLS, March 2009. Online: http://tools.ietf.org/html/draft-rescorla-tls-extended-random-02. Internet-Draft version 02.

[26] RSA Security Inc. The RSA watermark. 2009. Online: https://developer-content.emc.com/docs/rsashare/share_for_java/1.1/dev_guide/group__LEARNJSSE__WATERMARK.html.

[27] Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. Cryptology ePrint Archive, Report 2006/190, 2006. Online: http://eprint.iacr.org/.

[28] Dan Shumow and Niels Ferguson. On the possibility of a back door in the NIST SP800-90 Dual Ec Prng. CRYPTO 2007 Rump Session, August 2007. Online: http://rump2007.cr.yp.to/15-shumow.pdf.

[29] Sebastian Zander and Steven J. Murdoch. An improved clock-skew measurement technique for revealing hidden services. In *Proceedings USENIX Security 2008*, pages 211–226, 2008. Online: http://www.cl.cam.ac.uk/~sjm217/papers/usenix08clockskew.pdf.

iSeeYou: Disabling the MacBook Webcam Indicator LED

Matthew Brocker
Johns Hopkins University

Stephen Checkoway
Johns Hopkins University

Abstract

The ubiquitous webcam indicator LED is an important privacy feature which provides a visual cue that the camera is turned on. We describe how to disable the LED on a class of Apple internal iSight webcams used in some versions of MacBook laptops and iMac desktops. This enables video to be captured without any visual indication to the user and can be accomplished entirely in user space by an unprivileged (non-root) application.

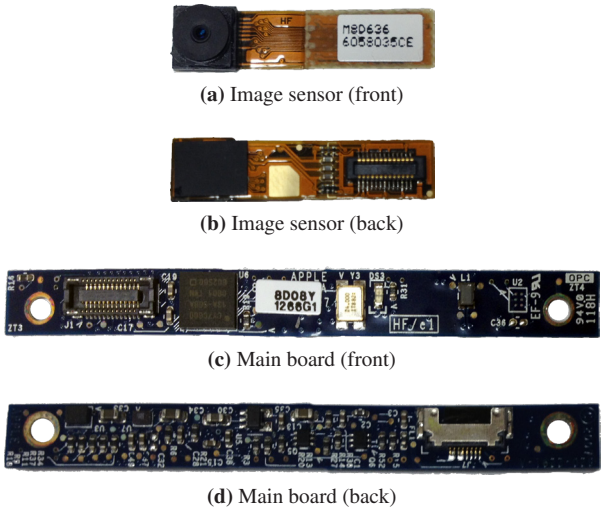
The same technique that allows us to disable the LED, namely reprogramming the firmware that runs on the iSight, enables a virtual machine escape whereby malware running inside a virtual machine reprograms the camera to act as a USB Human Interface Device (HID) keyboard which executes code in the host operating system.

We build two proofs-of-concept: (1) an OS X application, *iSeeYou*, which demonstrates capturing video with the LED disabled; and (2) a virtual machine escape that launches *Terminal.app* and runs shell commands. To defend against these and related threats, we build an OS X kernel extension, *iSightDefender*, which prohibits the modification of the iSight's firmware from user space.

1 Introduction

Video is ineffably compelling. The (consensual) sharing of video is an act of intimacy as it allows the viewer a glimpse into the life of the sharer. It is no surprise then that the Internet's first "lifecast," Jennifer Ringley's "JenniCam" in 1996 [24], was video and not audio. Similarly, YouTube, the most popular website for sharing user-created videos, predates SoundCloud, a website with similar functionality for audio, by several years even though technological constraints would suggest the opposite order. It is precisely because of the intimacy of video that turning on someone's camera without his or her knowledge or consent is a violation more fundamental than recording audio.

Beyond intentional sharing, video makes for more compelling evidence that an event occurred as claimed than either an after-the-fact eye witness account or audio recording. This is true whether it is a video of a successfully performed feat of skill — e.g., in sports [44] or even video games [49] — video of police brutality [55], video of violent crime [63], or webcam video used for blackmail [15].



(a) Image sensor (front)

(b) Image sensor (back)

(c) Main board (front)

(d) Main board (back)

Figure 1: The iSight from a 2008 MacBook we studied.

The value of video evidence is so high that The Washington Post recently reported that the US Federal Bureau of Investigation (FBI), has developed surveillance malware, similar to the proof-of-concept described in this paper, which can covertly turn on a victim's webcam [59]. Of course, the threat to privacy from webcams vulnerable to hacking comes not only from law enforcement.

At the beginning of the 2008 school year, the Lower Merion School District provided a MacBook laptop to each enrolled student. These laptops came pre-loaded with the *LANrev* remote administration tool (RAT) which allowed school district officials to, among other things, capture images from the MacBooks' built-in iSight webcam. During the following 18 months, officials captured more than 30 thousand images from these webcams [5, 6]. The first indication that images were being captured was every time the software took a picture, the green indicator LED would briefly illuminate [5, 6, 42]. Some teachers were so concerned by this they they covered the lens of the webcams on their own laptops [6]. Here, the indicator LED worked exactly as it was supposed to and alerted the users that they were being photographed.

The possibility that a webcam could be capturing pictures without the LED illuminating has led to suggestions that owners should tape over the webcam [43] as well as products designed to cover the camera stickers [10, 58].

This incident illustrates the dangers of passive sensors attached to computers like cameras, microphones, and GPS receivers. Unlike active input devices like keyboards and mice that require user actions to provide input, a passive sensor requires no action on the part of the user to capture input. Indeed, a user is typically unaware that input is being captured at all unless specific mechanisms are built into the technology to indicate that the sensor is currently in use. Such mechanisms include camera-use indicator LEDs, shutter sounds on cell phone cameras, and GPS-use indicator icons on mobile devices and laptops.

In the past few years, the ever-expanding set of sensors present in commodity laptops and smart phones has prompted the security and privacy community to begin researching ways to detect and limit the undesired use of sensors [20, 22, 26, 27, 31]. At the same time, researchers have demonstrated attacks exploiting the presence of sensors such as a clickjacking attacks against Adobe Flash to gain access to the camera and microphone [23] from a malicious web page and exfiltrating audio from microphones in modern automobiles [11]. (See Section 2 for more examples.)

Our results in this paper demonstrate that, at least in some cases, people have been correct to worry about malware covertly capturing images and video. We show a vulnerability in the iSight webcam that affects a particular range of Apple computers — including the MacBooks given to the students in the Lower Merion School District — that can be exploited to turn on the camera and capture images and video without the indicator illuminating.

At a high level, our investigation of the iSight revealed that it is designed around a microprocessor and a separate image sensor with an indicator LED sitting between them such that whenever the image sensor is transmitting images to the microcontroller, a hardware interlock illuminates the LED. We show how to reprogram the microcontroller with arbitrary, new firmware. This in turn enables us to reconfigure the image sensor, allowing us to bypass the hardware interlock and disable the LED. We also show a new method of performing a virtual machine escape based on our ability to reprogram the microcontroller.

Specifically, our technical contributions in this paper are five-fold:

1. We describe the architecture of the Apple internal iSight webcam found in previous generation Apple products including the iMac G5 and early Intel-based iMacs, MacBooks, and MacBook Pros until roughly 2008 (Section 3).
2. We demonstrate how to bypass the hardware interlock that the iSight uses to turn on the indicator LED whenever the camera is capturing images or video (Section 4) and provide a proof-of-concept user space application, *iSeeYou*, to do so (Section 6).
3. We demonstrate how to use the capability developed to bypass the hardware interlock to achieve a virtual machine escape (Appendix A[1]).
4. We develop an OS X kernel extension, *iSightDefender*, to defend against these attacks (Section 7).
5. We sketch the design space for building a secure camera module (Section 8).

The ability to bypass the interlock raises serious privacy concerns and the technical means by which we accomplish it raises additional security concerns which we discuss in Section 9.

Threat model. To mount our main attack where we capture video without any external indication to the victim, we assume that an attacker is able to run native code on the victim's computer as an unprivileged user. Further, we assume the code is unencumbered by defenses such as Apple's *App Sandbox* [4] which is used for applications downloaded from the Mac App Store but by little else. This assumption is quite mild and would typically be satisfied by malware such as RATs.

For the virtual machine escape, we assume the attacker has code running locally in the virtual machine and with whatever privileges the guest OS requires to communicate with USB devices. We also assume that the virtual machine monitor has exposed the iSight device to the virtual machine. This second assumption is quite strong as virtual machine monitors typically do not expose USB devices to the guest OS unless the user specifically configures it to do so, for example to use video conferencing software.

Generality of results. We stress that our main result — disabling the iSight LED — only applies to the first generation internal iSight webcams, found in some Apple laptops and desktops, and we make no claims of security or insecurity of later models, including the most recent (renamed) FaceTime cameras. The virtual machine escape described in Appendix A likely holds for other USB devices that use the Cypress EZ-USB chip used in the iSight, but we have not yet tested other devices.

2 Related work

General purpose computers contain a variety of processors designed for performing specialized tasks other than general-purpose computation. Examples include graphics processing units (GPUs) which produce video output; processors in network interface controllers (NICs) which perform network packet processing; microcontrollers in perhipherals such as keyboards, mice, and webcams; microcontrollers in laptop batteries; and, in some systems, baseboard management controllers (BMCs) which en-

[1] Although we regard this as a major contribution, we have moved the details to an appendix to improve the paper's flow

ables out-of-band system management independent of the host computer's CPU.

Security researchers have only recently begun examining these additional processors and the firmware that runs on them. In many cases, the designers of these systems appear not to have appreciated the security implications of their interfaces and implementations.

Perhaps the most well-studied processor apart from the CPU is the GPU. Vasiliadis et al. [60] demonstrate using the GPU to harden malware against detection by using the GPU to implement unpacking and runtime polymorphism. Ladakis et al. [33] use the GPU's direct memory access (DMA) capability to monitor the system's keyboard buffer to build a keylogger. Beyond GPU malware itself, researchers have used the GPU to accelerate malware detection [32] and intrusion detection systems [50].

Duflot and Perez [17] demonstrate exploiting a NIC to achieve arbitrary code execution. In follow up work, Duflot et al. [18] build a NIC malware detection framework.

Miller [39] demonstrates how to communicate with Apple laptop batteries using the System Management Bus, authenticate to the battery to "unseal" it, and change both configuration values and firmware. This enables overcharging the battery resulting in overheating and, potentially, leading to a fire.

Tereshkin and Wojtczuk [57] introduce the concept of a "Ring −3" rootkit which runs on Intel's Active Management Technology (AMT) hardware which has a processor independent of the host CPU with a separate interface to the NIC and DMA access to main memory.

In a very similar vein, Farmer [21] discusses weaknesses and vulnerabilities in the Intelligent Platform Management Interface (IPMI) — the standard interface to the baseboard management controller (BMC). Like AMT, a BMC has direct access to the host system but its operation is completely independent making exploits both extremely powerful and difficult to detect. Moore [41] builds on this work to produce a penetration tester's guide for examining IPMI and BMCs.

A webcam is just a particular type of sensor attached to a computing device. Others include microphones, accelerometers, and GPS receivers. Our work joins an emerging line of research on the security and privacy implications of such sensors. For example, Schlegel et al. [54] show how to use a smartphone's microphone to extract credit card numbers and PINs from spoken and tone-based interfaces. Marquardt et al. [36], Owusu et al. [46] and Miluzzo et al. [40] use smartphone accelerometers to extract information about key presses. Checkoway et al. [11] extract audio and GPS coordinates from automobiles. Templeman et al. [56] use smartphone cameras to covertly take pictures which are then used to create 3D models of physical spaces.

Our virtual machine escape (Appendix A) is not the first to emulate a USB Human Interface Device (HID) such as a mouse or keyboard. Wang and Stavrou [62] use a compromised smart phone to act as a USB HID keyboard and send key presses to the host system. Kennedy and Kelley [30] use a small microcontroller to interact with the Windows Powershell. Pisani et al. [48] similarly describe having USB devices pose as HID keyboards to control the computer. Elkins [19] adds a RF receiver for remote controlling a fake HID keyboard.

3 Internal iSight architecture

This section describes the architecture of the internal iSight webcam in sufficient detail to understand how the multi-step attack described in Section 4 works. Readers who are already familiar with the iSight or the Cypress EZ-USB or who are not interested in the low-level details of the device are encouraged to skip directly to Section 4 and use this section and Figure 2, in particular, as a reference as needed.

The internal iSight consists of a Cypress CY7C68013A EZ-USB FX2LP, a Micron MT9V112 CMOS digital image sensor, a 16 byte configuration EEPROM, and an indicator LED (see Figure 1). A block diagram is given in Figure 2.

3.1 Cypress EZ-USB

The host computer interacts with the iSight entirely through a USB connection to the Cypress EZ-USB. The EZ-USB is responsible for handling all USB requests and sending replies including video data.

The EZ-USB has an internal Intel 8051-compatible microcontroller core and 16 kB of on-chip RAM accessible as both code and data "main" memory[2] but lacks persistent storage [13]. In general, the firmware for the 8051 core can be located in one of three locations: (1) external memory such as flash or EPROM attached to the EZ-USB address/data bus; (2) an I^2C EEPROM; or (3) loaded from USB. The iSight loads its firmware at boot from the host computer over USB (see Section 4.2).

3.2 Micron digital image sensor

The Micron digital image sensor is a low-power system-on-a-chip (SOC) capable of producing an image in several formats. The sensor is configured by the I^2C interface which can read from and write to several hundred configuration registers [37]. In addition to the I^2C interface, several hardware signals influence the operation of sensor.

The most important signals from our perspective are the active-low `#RESET` and active-high `STANDBY` sig-

[2] The standard 8051 is a Harvard architecture which has separate code and data memory differentiated by hardware signals. In the configuration used by the iSight, the signals are combined effectively giving a single main memory address space.

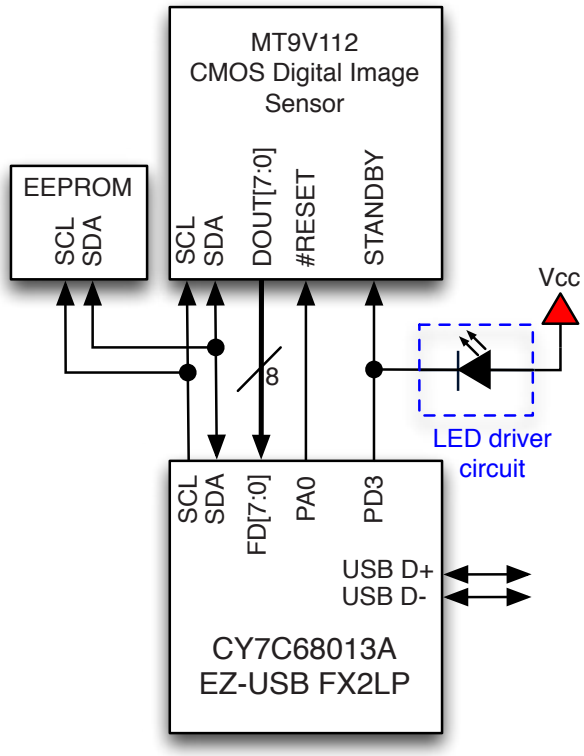


Figure 2: Internal iSight architecture block diagram consisting of a Cypress EZ-USB, a Micron digital image sensor, a 16 byte configuration EEPROM, and an indicator LED. The SCL and SCA lines comprise the I^2C bus.

nals. The corresponding hardware pins are connected directly to the EZ-USB's general purpose I/O (GPIO) pins. As shown in Figure 2, `#RESET` is connected to pin 0 of GPIO port A and `STANDBY` is connected to pin 3 of GPIO port D. The other connection between the image sensor and the EZ-USB shown in Figure 2 `DOUT[7:0]`→`FD[7:0]` is an 8 bit unidirectional bus which transfers pixel data to the EZ-USB's FIFO interface. Other, less important, control signals are omitted from the diagram.

The `#RESET` signal performs a hardware reset, resetting all configuration registers to their default value. The `STANDBY` signal controls output enable and power down functions. That is, when `STANDBY` is asserted, the image sensor stops producing data on `DOUT[7:0]` which enters the high impedance state as well as allowing the image sensor to transition to a low-power state.

3.3 Configuration EEPROM

The first byte of the 16 byte EEPROM controls whether the EZ-USB loads its firmware from USB or from the EEPROM itself. When set to load firmware from USB, as the iSight does, the EEPROM contains the USB vendor ID (VID), product ID (PID), device release number, and a configuration byte for the initial device enumeration. Once the EZ-USB has enumerated using the VID, PID, and release values, software on the host computer can load the firmware. The iSight initially enumerates with vendor ID `0x05ac` (Apple, Inc.) and product ID `0x8300` (Built-in iSight (no firmware loaded)).

Table 1: Relation between the `PD3` GPIO, the `STANDBY` signal, and the LED.

PD3	STANDBY	LED
High	Asserted	Off
Low	Deasserted	On

3.4 Indicator LED

Since the purpose of the indicator LED is to illuminate whenever the camera is capturing video, a LED driver circuit is connected directly to the `STANDBY` input of the image sensor (see Figure 2). In this way, whenever `PD3` is high — that is, `STANDBY` is asserted — the LED is off and whenever `PD3` is low — so `STANDBY` is deasserted and the image sensor is producing output — the LED is on. Since the LED is controlled by the same output that controls `STANDBY`, there is no danger that firmware on the EZ-USB could deassert `STANDBY` *and* turn the LED off (see Table 1). However, as we demonstrate in Section 4, we can bypass the `STANDBY` signal such that changing `PD3` allows us to control the LED without affecting the operation of the image sensor.

4 Disabling the indicator LED

Disabling the indicator LED on the iSight entails two requirements. First, as described in Section 3, the indicator LED is directly connected to the `STANDBY` pin on the image sensor. In order to disable the LED, we need to keep `STANDBY` asserted. Since asserting `STANDBY` will disable the image sensor output, we need to configure the image sensor to ignore `STANDBY` before we assert this signal. Second, we need a way to modify the firmware on the EZ-USB to in order to configure the image sensor appropriately as well as keep `STANDBY` asserted whenever we want the LED to stay off.

4.1 Bypassing the `STANDBY` signal

The Micron image sensor has a 16 bit configuration register, RESET (which is distinct from the `#RESET` power-on-reset signal). RESET is addressable from the I^2C interface at address `0x0D` in register page 0 [37]. The most significant 8 bits control hardware clocks and how bad frames should be handled which are of no interest to us and can be left as 0. The least significant 8 bits have the following functionality as described in the image sensor

data sheet [37, Table 13]:

- Bit 7. Prevent `STANDBY` from affecting entry to or exit from the low-power state if set.
- Bit 6. Prevent `STANDBY` from contributing to output enable control if set.
- Bit 5. Reset the SOC (but not the sensor) if set.
- Bit 4. Disable pixel data output if set.
- Bit 3. Chip enable. Normal operation if set, no sensor readout otherwise.
- Bit 2. Software standby if set, otherwise normal operation.
- Bit 1. Restart reading an image frame.
- Bit 0. Reset the sensor to its default state if set, normal operation otherwise.

Bits 0, 1, and 5 are of no interest and can be set to 0 but the remaining 5 bits enable us to bypass the `STANDBY` signal while still maintaining normal operation. This includes entering a (software) standby state and disabling output when appropriate.

When the iSight is first powered up (or, more precisely, when `#RESET` becomes deasserted), the RESET register has value `0x0008`; that is, normal operation and `STANDBY` affects the low-power state and output enable. If RESET is set to `0x00c8`, then the camera has normal operation but `STANDBY` is effectively bypassed. When it becomes desirable for the camera to enter the standby state, RESET can be set to `0x00d4` which disables output and enters the software standby state.

With RESET set to either `0x00c8` or `0x00d4`, the hardware `STANDBY` signal is ignored. This enables the use of the EZ-USB `PD3` output to control the LED independent of the standby state of the image sensor.

4.2 Programming the EZ-USB

When the iSight is first powered, it checks the configuration EEPROM and then waits for programming over USB (see Section 3.3). The *AppleUSBVideoSupport* I/O Kit driver matches the vendor ID (VID) and product ID (PID). The driver loads and the `AppleUSBCamera::start()` function downloads the camera's firmware (stored in the `gTheFirmware` array) to the EZ-USB using a series of vendor-specific USB "Firmware Load" device requests [13, Section 3.8]. The camera will then reenumerate and function as a webcam.

One approach to change the firmware on the camera is to modify the *AppleUSBVideoSupport* driver to contain different firmware. A second approach would be to provide a new driver that matches the VID/PID and provides a higher probe score [2]. The new driver would run at system start up instead of Apple's driver and download the new firmware to the camera. These approaches have two major drawbacks. The first drawback is that they rely on programming the iSight when it is in its unprogrammed state which only happens when the camera is first powered by the USB bus. The second drawback is that root access is required in order to modify the existing driver or load a new driver.

A third approach overcomes both drawbacks by letting the iSight be programmed with the legitimate firmware when it is first powered. Once the firmware has been loaded onto the camera, it can be reprogrammed at any time using "Firmware Load" requests. Furthermore, it can be reprogrammed from any user space process.

5 Finding the vulnerability

The information described in Sections 3 and 4 was discovered by a combination of reverse engineering, experimentation, and reading data sheets once individual components were identified. We started by ordering camera modules from a variety of Apple computers on eBay. Coincidentally, the modules were all from the original iSight camera, although the camera boards for the MacBook and iMac had different forms. Figure 1 shows the MacBook board.

A cursory examination of the board reveals that the camera microprocessor is a Cypress EZ-USB. The EZ-USB Technical Reference Manual [13] describes the procedure to download code to EZ-USB. We reverse engineered the *AppleUSBVideoSupport* driver using IDA [25] to determine the format of the firmware stored in the driver. (Section 6.1 describes the firmware in more detail.) We then extracted the firmware as it would appear in memory and analyzed it using IDA.

Our initial hypothesis was that the LED would be controlled by one of the EZ-USB GPIO pins via the firmware. To test this, we mapped out the connections on the board using a digital multimeter with a specific focus on connections from the microcontroller to the indicator LED. A connection was found between the microcontroller, image sensor, and the LED driver circuit. Since the microcontroller pin connected to the LED was set as an output, we constructed new firmware to toggle this output and examined the results. When the LED was turned on, the camera functioned correctly. When the LED was turned off, the camera ceased operating (see Table 1).

Since the output controlling the LED was also connected to the image sensor, we examined it next. When the legitimate camera firmware is downloaded to the camera, it identifies itself as "Apple, Inc. Built-in iSight [Micron]" suggesting that the image sensor was manufactured by Micron Technology. There is no visible part number that can be used to identify the model (see Figure 1). Rather than decapping the chip, we used the Wayback Machine[3] to view the Micron website for 2005, the year the camera board was copyrighted. Data sheets for the

[3] `https://archive.org/web/`

image sensors that matched the publicly known specs for the iSight camera on Micron's website indicate that the image sensor communicates over an I^2C bus. One of the I^2C-addressable registers identifies the chip version. We identified the I^2C bus and read the register which revealed the particular image sensor.

We examined the relevant data sheet for the image sensor and noticed the `STANDBY` pin with functionality consistent with our experiments toggling the LED-controlling output pin. After reading the data sheet in more detail, we discovered the I^2C-addressable register which enables a software override for the `STANDBY` pin. Further experiments with modified firmware were performed to verify that the LED driver circuit was indeed connected to `STANDBY` and that it could be bypassed.

6 Proof of concept

The discussion in Section 4 shows that, in principle, it is possible to modify the legitimate firmware to disable the LED. In this section, we describe the proof-of-concept application, *iSeeYou* we created which reprograms the iSight to add the capability to enable or disable the LED using a new vendor-specific USB device request.

6.1 Modifying the firmware

Although one could reimplement the camera functionality, we opted to create new firmware by appending new binary code to the legitimate firmware and patching it to call our new code. The first step is to extract the legitimate firmware from the *AppleUSBVideoSupport* device driver.[4]

The firmware consists of an 8 byte header followed by a sequence of triples: a 2 byte size, a 2 byte address, and size-bytes of data. This format corresponds exactly to the "C2 Load" format of the EEPROM for loading firmware directly from the EEPROM [13, Table 3-6]. Each triple specifies the data that should be written to the EZ-USB's main memory at a given address. By stripping off the header and the final triple,[5] we can construct the "raw" firmware image. The raw firmware can then be analyzed using IDA.

The raw firmware is structured similarly to sample code provided in the Cypress EZ-USB FX2LP Development Kit [14] including a hardware initialization function and USB events that are serviced by a main loop based on state bits set by interrupt handlers.

To the legitimate firmware, we add two bits of state, "is the sensor in software standby or running" and "is the LED enabled or disabled," as well as four new functions, `reset_sensor`, `enter_standby`, `exit_standby`, and `handle_led_control`.

When the LED is enabled, the behavior of the camera is indistinguishable from the normal behavior. That is, when the camera is in its standby state the LED is off and when the camera is in its running state, the LED is on.

The legitimate firmware contains a function to reset and configure the image sensor. This is called both from the hardware initialization function and the handler for the USB set interface request. It begins by deasserting the `STANDBY` signal and asserting the `#RESET`. After a short spin loop, it deasserts `#RESET` and, depending on the function argument, deasserts `STANDBY`. It then proceeds to configure the image sensor. We patch the firmware to call `reset_sensor` instead of this configuration function in both locations. The `reset_sensor` function reimplements the reset functionality but adds a call to the function which writes to the I^2C bus to program the RESET register to bypass the `STANDBY` signal (see Section 4.1). At this point, if the LED has been disabled or the argument indicates that it should enter the standby state, the `STANDBY` signal is asserted to turn off the LED which will have momentarily illuminated during the reset sequence. Otherwise, the sensor is left running and the LED is enabled so `STANDBY` remains deasserted and the LED stays on. Finally, the `reset_sensor` function jumps into the middle of the configuration function, just past the `#RESET` and `STANDBY` manipulating code, in order to perform the rest of the configuration.

The `enter_standby` and `exit_standby` functions update the bit of state which records if the image sensor is running or in standby. Then, based on whether the LED is enabled or not, they deassert (resp. assert) `STANDBY` as needed to turn the LED on (resp. off). Finally, these functions use I^2C to program the RESET register to enter or exit software standby. Each location in the legitimate firmware which sets the state of the `STANDBY` signal is patched to call its new, corresponding standby function instead.

The final function, `handle_led_control` is responsible for handling a new vendor-specific USB device request. The main loop in the legitimate firmware which handles USB device request "setup" packets is patched to instead call `handle_led_control`. If the `bRequest` field of the request does not match the new vendor-specific value, then it jumps to the legitimate handler. Otherwise, based on the `wValue` field of the request, the LED is enabled or disabled. As with the other functions, the LED is then turned on if it has been enabled and the image sensor is running. Otherwise, it is turned off.

6.2 Demonstration application: *iSeeYou*

iSeeYou is a simple, native OS X application; see Figure 3. When *iSeeYou* starts, it checks for the presence of

[4] There are several open source tools to perform this task, e.g., iSight Firmware Tools [7], several of which include binary patching to fix bugs in the USB interface descriptors.

[5] The final triple stores a single `0x00` byte to address `0xE600` which takes the Intel 8051 core out of reset so that it can begin executing instructions.

Figure 3: *iSeeYou* running on a white MacBook "Core 2 Duo" capturing video from the internal iSight with the LED (the black dot to the right of the square camera at the top, center of the display bezel) unilluminated.

a built-in iSight using the appropriate vendor and product IDs. If the iSight is found, *iSeeYou* initiates the reprogramming process using the modified firmware described above. Once the camera has been reprogrammed and has reenumerated, the start/stop button begins/ends capturing and displaying video. The `LED Enable`/`LED Disable` control sends USB device requests with the new vendor-specific value to enable/disable the indicator LED while video is being captured. Finally, when the user quits *iSeeYou*, the camera is reprogrammed with the legitimate firmware.

7 Defenses

There are several approaches one can envision to defend the iSight against the attacks described in the previous sections. One can change (1) the hardware, (2) the firmware on the EZ-USB (unfortunately this is not effective, see below), or (3) the software on the host system. See Table 2 for an overview of possible defenses and their efficacy.

The most comprehensive defense would be to change the hardware used in the iSight. See Section 8 for several secure hardware designs. Of course, changing the hardware is not a deployable solution for existing devices.

Table 2: Overview of possible defenses.

Defense	Deployable	User	Root
Change hardware	No	Yes	Yes
Change firmware	Yes	No	No
App Sandbox	Yes	Some	No
iSightDefender	Yes	Yes	No

A "Yes" in the Deployable column indicates that the defense could be deployed to existing computers. A "Yes" in the User (resp. Root) column indicates that the defense would prevent an unprivileged (resp. root) process from reprogramming the iSight. A "Some" indicates that some reprogramming attempts would be prevented but others allowed.

If the hardware must remain the same, then if the firmware on the camera could be changed to disallow future reprogramming, then the camera would be secure against our attacks. Unfortunately, the "Firmware Load" USB device request used to reprogram the 8051 core is handled entirely by the EZ-USB device itself and cannot be blocked or handled by the 8051 itself [13, Section 3.8].

Thus no matter how one programs the device's firmware, it can be reprogrammed by an attacker who can send basic USB messages to the camera.

Apple deploys sandboxing technology called the *App Sandbox*[6] [4] which can prevent applications inside the sandbox from accessing the iSight. Specifically, the `com.apple.security.device.camera` entitlement enables an application to capture still images and video from cameras, including the internal iSight. The `com.apple.security.device.usb` entitlement enables applications to access USB devices.

Any *App Sandbox*–protected application lacking the `usb` entitlement would be prohibited from reprogramming the iSight and thus prohibited from disabling the indicator LED. Although an application with the `usb` entitlement but lacking the `camera` entitlement would be prohibited from using the high-level APIs for accessing the camera, such as the QTKit API [3], it could easily reprogram the camera to not appear as a USB video class (UVC) device and instead transfer the frames of video using a custom protocol.

The major drawback to using the *App Sandbox* to protect the camera is that applications need to opt into the protection, something malware is unlikely to do. Worse, the *App Sandbox* has, at times, been broken allowing applications to escape from the restrictions [12, 38].

Perhaps the best way to defend against reprogramming the iSight without changing the hardware is to modify the operating system to prevent particular USB device requests from being sent to the camera. We have built such a defense structured as an OS X kernel extension called *iSightDefender*.

When iSight is powered for the first time, it enumerates with vendor ID `0x05ac` and product ID `0x8300` and is programmed with the legitimate firmware via the *AppleUSBVideoSupport* kernel extension as described in Sections 3.3 and 4.2. When it reenumerates with product ID `0x8501` the kernel matches and loads the normal drivers as well as *iSightDefender*.

I/O Kit kernel drivers are written in a subset of C++ and each USB device is represented by an object of class `IOUSBDevice` which is responsible for communicating with the hardware by sending messages to objects in lower layers of the USB stack. When *iSightDefender* is started, it overwrites the C++ virtual method table of its "provider" `IOUSBDevice` to point to the virtual method table of a subclass of `IOUSBDevice`.[7] The subclass overrides the four `DeviceRequest` member functions. The overridden implementations check if the device request is for the "Firmware Load" vendor-specific request and, if so, log the attempt in the system log and block the request.

iSightDefender is able to block all user space reprogramming attempts,[8] including those mounted from within a virtual machine. The latter requires some care as the normal drivers that match against the `IOUSBDevice` are unloaded and the virtual machine monitor's own driver is loaded in their place.

Using *iSightDefender* raises the bar for attackers by requiring the attacker to have root privileges in order to reprogram the iSight. In some sense, this is the strongest possible software-based defense. Since malware running as root would have the ability to replace or modify kernel code, any defense implemented in the kernel can, theoretically, be bypassed. Despite this limitation, we believe it is a step in the right direction and encourage its use.

iSightDefender, and its source code, is freely available.[9]

8 Secure camera designs

When designing a secure camera, there are two main considerations. First, for sensors such as cameras and microphones, an indicator that the sensor is recording is essential to prevent surreptitious recording. (Although laptop microphones do not, in general, have indicators, it is common for stand alone USB microphones; see [29] for an example.) For the highest level of assurance that the indicator cannot be bypassed, the indicator should be controlled completely by hardware.

Second, as with any peripheral connected to the computer, a vulnerability in the firmware running on the peripheral or the ability to reprogram the firmware enables an attacker to leverage all of the capabilities of the peripheral. Section 2 contains numerous examples of this. The virtual machine escape in Appendix A is another example where an attacker leverages the USB connection and the ability of the EZ-USB to mimic any USB device to the host computer. Apple's most recent FaceTime cameras in its 2013 MacBook Air model eschews USB 2.0. Instead, the camera is connected to the host computer over PCIe [35]. Vulnerabilities in the camera would potentially enable an attacker to have DMA access to the host system. This is a significantly stronger capability than USB access.

8.1 Secure indicators

Laptop cameras are typically constructed by pairing a CMOS image-sensor-on-a-chip (e.g., the Mi-

[6] Formerly codenamed *Seatbelt*.

[7] There seems to be no supported mechanism for interposing on USB device requests. The authors appreciate the irony of using virtual table hijacking — a common hacker technique — for defending against attack.

[8] In fact, *iSightDefender* worked so well that one author spent more than an hour attempting to diagnose (nonexistent) problems with *iSeeYou* before noticing the tell-tale lines in the system log indicating that *iSightDefender* had been loaded by a computer restart and it was blocking reprogramming requests.

[9] `https://github.com/stevecheckoway/iSightDefender`

cron MT9V112 found in the iSight or the Toshiba TCM8230MB(A)) with a separate microcontroller that handles communication with the host computer (e.g., the EZ-USB FX2LP found in the older MacBooks or the Vimicro VC0358 [61] found in more recent MacBook Pros [28]. There are, of course, many possible combinations of image sensors and microcontrollers one could use.

Image-sensors-on-a-chip tend to have a number of common features that can be used to build a secure indicator.

1. Separate power connection for CMOS sensor itself. For example, `VAAPIX` on the MT9V112 powers its pixel array and `PVDD` on the TCM8230MB(A) powers its photo diode. A GPIO pin on the microcontroller can be connected to both the LED driver circuit and the CMOS sensor power supply circuit. Whenever images are to be captured, the microcontroller sets its GPIO pin appropriately, power is supplied to the sensor and the LED turns on.
2. `#RESET` pins. The LED driver circuit can be connected to the `#RESET` pin and a GPIO pin on the microcontroller. The microcontroller would hold the image sensor in reset whenever it was not capturing images. Compared to the power connection for CMOS sensor, holding the entire sensor-on-a-chip in reset means that before images could be captured, the sensor would need to be reconfigured. Reconfiguring typically means sending a few dozen bytes over an I^2C or SPI bus. This introduces a slight delay.
3. Output clocks and synchronization signals. Image sensors typically latch outputs on one edge of an output clock signal and image consumers are expected to read the data on the other edge of the clock. In addition, there are signals used to indicate which part of the image the latched data represents. For example, the MT9V112 has `FRAME_VALID` and `LINE_VALID` signals indicating when it's outputting a frame or a line within the frame, respectively, whereas the TCM8230MB(A) has `VD` and `HD` for vertical and horizontal synchronization. These pins can also be used to control the LED by adding some simple hardware that drives the LED if it has seen one of these signals change in the past few milliseconds.

   Depending on the specifics of the image sensor output signal, a retriggerable, monostable multivibrator can be used to drive the LED as long as its input changes sufficiently often. The multivibrator's output pulse width needs to be set appropriately such that it is triggered frequently enough to continuously drive the LED while images are being recorded.

   Some care must be taken when using these output signals. The exact meanings of the signals can frequently be changed by configuring the sensor. This is analogous to the situation with the iSight where we changed the meaning of the `STANDBY` signal.

An all-in-one design where the image sensor is integrated with the microcontroller which communicates to the host computer is likely to have fewer options for a secure design. A dedicated output pin which could drive an indicator LED would suffice. However, hardware designers are typically loathe to dedicate pins to specific functions, instead a variety of functions tend to be multiplexed over a single pin.

It is likely that, even in this case, there would be a separate power connection for the CMOS sensor. As with the two-chip design above, the LED driver circuit and a power supply circuit could be driven by a GPIO.

8.2 Secure firmware

Although using one of the secure indicator designs described above will ensure the LED will turn on when the camera turns on, it does nothing to protect against reprogramming attacks.

For this, we make four concrete recommendations which, taken together, can secure the firmware on the camera. These apply more generally to any peripheral or embedded system connected to a host computer.

1. Store the firmware in nonvolatile storage on the camera module. Most commercial off-the-self (COTS) microcontrollers contain some amount of nonvolatile storage, such as flash memory, to hold firmware.[10] By programming the firmware at the factory, one avoids the possibility that the legitimate firmware will be replaced by an attacker on the host system before being downloaded to the microcontroller.

   Depending on the specific requirements of the system, the factory programming could be the complete firmware or a secure loader designed to load cryptographically signed firmware from the host (see below).
2. Use a microcontroller which can block unwanted firmware reprogramming attempts. It is essential that trusted code running on the microcontroller is able to block reprogramming attempts for illegitimate firmware.
3. Firmware updates, if necessary, should be cryptographically signed and the signature verified before applying the update. This requires both nonvolatile storage for the code to verify the signature and a microcontroller which can block reprogramming attempts. Since microcontrollers are typically resource constrained devices, choosing an appropriate signature scheme which can be implemented within the

[10] Microcontrollers without nonvolatile storage can be paired with external nonvolatile storage, such as flash or an EEPROM, to the same effect.

constraints is important. Scheme selection is outside the scope of this paper but we note that recent microcontrollers have started to contain specialized crypto instructions which can reduce code size and increase efficiency. For example, Rohde et al. [53] use specialized AES instructions in some Atmel ATxmega microcontrollers to implement the Merkle signature scheme.

4. Require root/administrator privileges to send reprogramming requests. Strictly as a matter of defense in depth, software running on the host system should restrict reprogramming attempts. Thus, even if the hardware and firmware defenses prove inadequate, this added layer of protection can still defend against some attacks.

   Adding this sort of restriction typically involves a device-specific kernel module (our *iSightDefender* is an example). This may be more difficult for plug and play devices expected to conform to standard protocols and interact with generic drivers such as USB video class (UVC) or USB human interface device (HID) class devices.

The inability of the EZ-USB to block reprogramming attempts indicates that this widely-used microcontroller is inappropriate for use in any system where security is a consideration.

Secure physical user interface Orthogonal to secure indicators and secure software is a secure physical user interface. Most webcams in laptops are controlled by software: Software tells the camera when to power up, when to capture video, and when to power down. A simple solution to the problem is to provide a physical switch similar to the switches found on laptop network adapters which controls power to the camera. A second simple solution is to provide a lens cover which the user must physically move aside to use the camera. This would be similar in spirit to the original external iSight and similar in form to the amusingly named iPatch [58].

9 Discussion

Although some webcams, such as the Logitech QuickCam Pro 9000, come with an explicit "LED control" that can disable the LED [64], such controls are not the norm and, in fact, are a very bad idea from both a security and a privacy stand point. Giving the user the ability to disable a privacy feature is tantamount to giving malware the same capability.

This work concerns the technical challenge of hardware exploitation; however, we would be remiss if we did not discuss the (frequently unpleasant) real-world consequences of vulnerabilities in privacy technology.

A particularly unsavory element of the hacker culture, "ratters," install malware bundled with remote administration tools (RATs) on victims' computers. There are several popular RATs, including Blackshades and DarkComet, which come with a variety of features such as keyloggers, the ability to install additional malware, and the ability to record video and sound using the webcam. Rats are often installed with the goal of spying on women.

RATs and the ratters who use them have recently come under public scrutiny after a recent Miss Teen USA's webcam was used by ratter Jared Abrahams to capture her naked images without her knowledge [15]. Abrahams arrest and guilty plea came on the heels of an ars technica exposé on ratters [1].

A commonly asked question on forums devoted to ratting, such as the *Hack Forums* "Remote Administrator Tools" forum, is how can one disable the webcam's LED. In one representative thread, forum user "Phisher Cat" asks

> So as all of you know, newer laptops have a light when a laptop webcam turns on, and so this scares the slave.
>
> Is it theoretically possible for a RAT to disable this light? [47]

disturbingly referring to his victim as "the slave," as is common in this subcommunity. The first response by "Jabaar" notes that "[p]eople have been trying to figure this out for a very long time. The light won't be able to be disabled as it is built into the hardware." Others agree: "Capital Steez" writes that there is "no way to disable it," and "FBI™" concurs "there [i]s no way to do" it. Still others suggest using social engineering in an attempt to convince the victim that the LED is normal, for example, "Orochimaru" writes, "You can't physically turn it off but you **can** use social engineering to fool them. Maybe send an error or warning msgbox that says 'Camera is now updating, please do not disturb' or something." There are many such threads on Hack Forums alone, all expressing similar sentiments: disabling LEDs is a capability the ratters *really* want to have but do not think is possible.

Unfortunately, the implications of surreptitiously capturing video do not end with privacy violations like law enforcement, school officials, and ratters spying on people. As part of the general trend of growing frustration with passwords as an authentication mechanism, some companies are moving to biometric identification; in particular, using facial recognition on video taken with webcams. For example, BioID is software-as-a-service which provides biometric identification to online service providers using a webcam [8]. Luxand's *FaceSDK* is a cross-platform software development kit that uses the webcam to identify the user [34].

In principle, this sort of facial recognition is trivially defeated by providing the expected picture or video to the software performing the authentication. Malware that

can capture video of the victim can replay that video to authenticate to the given service. This is not a new attack. The Android *Face Unlock* system was defeated shortly after being released by holding a picture of the face in front of the camera [9]. Duc and Minh [16] describes weaknesses of several facial recognition authentication systems in the presence of pictures. By disabling the indicator LED before capturing video, the victims have no way of knowing that their accounts may be compromised.

Although the ability to disable the LED can lead to serious privacy and security problems, there are at least two legitimate use cases. The first is that some people really do not want the LED on while they are recording. We do not find this to be a compelling use as the benefit does not seem to outweigh the potential cost; however, others may value this more than we do.

The second use case is significantly more compelling: laptop recovery. For example, the OS X version of *Adeona* software captures pictures using the laptop's internal iSight to aid in recovery of a laptop that has been stolen by taking a picture of the thief [51, 52]. The *LANrev* software used in the Lower Merion School District incident discussed in the introduction had a similar "Theft Track" feature which is how the school officials were able to obtain pictures of students. For this use, one does not want the thief to know he is being observed.

10 Responsible disclosure

The authors followed responsible disclosure practices by disclosing the LED disabling vulnerability to Apple product security team on July 16, 2013 and the virtual machine escape on August 1, 2013. The disclosures included the source code for *iSeeYou* and the virtual machine escape as well as directions for mounting both attacks. Apple employees followed up several times but did not inform us of any possible mitigation plans. The *iSightDefender* code was also provided to Apple and is now publicly available.[11]

11 Conclusions and future work

Engineering details of privacy technologies can have real-world consequences. As discussed in Sections 1 and 9, a computer user today potentially faces a variety of adversaries — from law enforcement and school officials to criminals — who want to capture images or video clandestinely. Currently, the only technological barrier standing in their way is the camera-on indicator LED. We have shown that, at least in some cases, the barrier can be overcome.

In particular, we have shown that being able to reprogram the iSight from user space is a powerful capability. Coupled with the hardware design flaw that allows the indicator LED hardware interlocks to be bypassed, malware is able to covertly capture video, either for spying purposes or as part of a broader scheme to break facial recognition authentication. Although the *iSightDefender* defense described in Section 7 raises the barrier for malware, including RATs, to take control of the camera without being detected by requiring root privileges, the correct way to prevent disabling the LED is a hardware solution.

In this paper, we have examined only a single generation of webcams produced by a single manufacturer. In future work, we plan to expand the scope of our investigation to include newer Apple webcams (such as their most recent high-definition FaceTime cameras) as well as webcams installed in other popular laptop brands.

The virtual machine escape described in Appendix A demonstrates the danger that reprogrammable peripheral devices such as keyboards and mice pose. We plan to undertake a much broader examination of these devices in an attempt to understand the security implications of connecting one device to a computer which can, under attacker control, pretend to be a wide range of devices. One particularly promising direction is to study how drivers react to malformed or malicious responses from devices. In the worst case, a user space program could reprogram a peripheral device which in turn exploits a poorly written driver to inject code into the kernel.

[11] *See supra* note 9.

Acknowledgments

We thank the anonymous reviewers for their detailed comments and helpful suggestions. We also thank Brian Kantor, Nick Landi, Eric Rescorla, Stefan Savage, Hovav Shacham, and Cynthia Taylor for many helpful discussions throughout this work and Kevin Mantey for letting us borrow test equipment and providing technical assistance.

References

[1] Nate Anderson. Meet the men who spy on women through their webcams. *ars technica*, March 2013. Online: `http://arstechnica.com/tech-policy/2013/03/rat-breeders-meet-the-men-who-spy-on-women-through-their-webcams/`.

[2] *I/O Kit Fundamentals: Driver and Device Matching*. Apple Inc., May 2007. Online: `https://developer.apple.com/library/mac/#documentation/devicedrivers/conceptual/IOKitFundamentals/Matching/Matching.html`.

[3] *QTKit Framework Reference*. Apple Inc., February 2009. Online: `http://developer.apple.com/library/mac/#documentation/QuickTime/`

Reference/QTCocoaObjCKit/_index.html.

[4] *App Sandbox Design Guide*. Apple Inc., March 2013. Online: http://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html.

[5] Ballard Spahr. Lower Merion School District forensics analysis: Initial LANrev system findings, May 2010. Online: http://www.scribd.com/doc/30891576/LMSD-Initial-LANrev-System-findings. Redacted.

[6] Ballard Spahr. Report of independent investigation: Regarding remote monitoring of student laptop computers by the Lower Merion School District, May 2010. Online: http://www.social-engineer.org/resources/100503_ballard_spahr_report.pdf.

[7] Étienne Bersac. iSight Firmware Tools. October 2009. Online: https://launchpad.net/isight-firmware-tools.

[8] BioID, Inc. The easy, secure way to log in and manage online identities and accounts. 2013. Online: http://mybioid.com/index.php?id=67. Last accessed: 2013-08-06.

[9] Matt Brian. Android 4.0 Face Unlock feature defeated using a photo. *The Next Web*, November 2011. Online: http://thenextweb.com/google/2011/11/11/android-4-0-face-unlock-feature-defeated-using-a-photo-video/.

[10] camJAMR.com. camJAMR.com webcam covers. 2012. Online: http://store.camjamr.com/shop-now/camjamr-webcam-covers.html. Last accessed: 2013-08-07.

[11] Stephen Checkoway, Damon McCoy, Danny Anderson, Brian Kantor, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In David Wagner, editor, *Proceedings of USENIX Security 2011*. USENIX, August 2011. Online: http://www.autosec.org/pubs/cars-usenixsec2011.pdf.

[12] CoreLabs, Core Security Technologies. Apple OS X Sandbox predefined profiles bypass. November 2011. Online: http://www.coresecurity.com/content/apple-osx-sandbox-bypass.

[13] *EZ-USB® Technical Reference Manual*. Cypress Semiconductor Corporation, 2011. Online: http://www.cypress.com/?docID=27095&dlm=1.

[14] Cypress Semiconductor Corporation. CY3684 EZ-USB FX2LP Development Kit. 2013. Online: http://www.cypress.com/?rID=14321.

[15] Alex Dobuzinskis. California man agrees to plead guilty to extortion of Miss Teen USA. *Reuters*, October 2013. Online: http://www.reuters.com/article/2013/10/31/us-usa-missteen-extortion-idUSBRE99U1G520131031.

[16] Nguyen Minh Duc and Bui Quang Minh. Your face is NOT your password: Face authentication bypassing Lenovo – Asus – Toshiba. Presented at BlackHat Briefings, July 2009. Online: https://www.blackhat.com/html/bh-usa-09/bh-us-09-main.html.

[17] Loïc Duflot and Yves-Alexis Perez. Can you still trust your network card? Presented at CanSecWest 2010, March 2010. Online: http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf.

[18] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Proceedings of RAID 2011*, pages 378–397. Springer, September 2011. Online: http://www.ssi.gouv.fr/IMG/pdf/paper.pdf.

[19] Monta Elkins. Hacking with hardware: Introducing the universal RF USB keyboard emulation device: URFUKED. Presented at DefCon 18, August 2010. Online: http://www.youtube.com/watch?v=EayD3V77dI4.

[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Remzi Arpaci-Dusseau and Brad Chen, editors, *Proceedings of OSDI 2010*. USENIX, October 2010. Online: http://static.usenix.org/events/osdi10/tech/full_papers/Enck.pdf.

[21] Dan Farmer. IPMI: Freight train to hell, January

2013. Online: http://fish2.com/ipmi/itrain.pdf.

[22] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In Stefan Katzenbeisser, Edgar Weippl, L. Jean Camp, Melanie Volkamer, Mike Reiter, and Xinwen Zhang, editors, *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*, pages 291–307. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-30921-2_17.

[23] Jeremiah Grossman. Clickjacking: Web pages can see and hear you. October 2008. Online: http://jeremiahgrossman.blogspot.com/2008/10/clickjacking-web-pages-can-see-and-hear.html.

[24] Hugh Hart. April 14, 1996: JenniCam starts lifecasting. *Wired Magazine*, April 2010. Online: http://www.wired.com/thisdayintech/2010/04/0414jennicam-launches/.

[25] Hex-Rays. IDA: About. January 2014. Online: https://www.hex-rays.com/products/ida/.

[26] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 'These aren't the droids you're looking for': Retrofitting Android to protect data from imperious applications. In George Danezis and Vitaly Shmatikov, editors, *Proceedings of CCS 2011*. ACM Press, October 2011. Online: https://research.microsoft.com/pubs/149596/AppFence.pdf.

[27] Jon Howell and Stuart Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In Collin Jackson, editor, *Proceedings of W2SP 2010*. IEEE Computer Society, May 2010. Online: https://research.microsoft.com/pubs/131132/devices-camera-ready.pdf.

[28] iFixit. MacBook Pro Retina Display teardown. 2013. Online: http://www.ifixit.com/Teardown/MacBook+Pro+Retina+Display+Teardown/9493.

[29] Samson Technologies Inc. Meteor Mic - USB Studio Microphone. 2013. Online: http://www.samsontech.com/samson/products/microphones/usb-microphones/meteormic/.

[30] David Kennedy and Josh Kelley. PowerShell omfg.... Presented at DefCon 18, August 2010. Online: http://www.youtube.com/watch?v=eWoAGxh0a_Q.

[31] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Stack analyzer for detecting privacy leaks in android applications. In Hao Chen and Larry Koved, editors, *Proceedings of MOST 2013*. IEEE Computer Society, May 2013. Online: http://www.mostconf.org/2012/papers/26.pdf.

[32] Nicholas S. Kovach. Accelerating malware detection via a graphics processing unit. Master's thesis, Air Force Institute of Technology, September 2010.

[33] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy GPU-based keylogger. In Thorsten Holz and Sotiris Ioannidis, editors, *Proceedings of EuroSec 2013*. ACM, April 2013. Online: http://www.cs.columbia.edu/~mikepo/papers/gpukeylogger.eurosec13.pdf.

[34] Luxland, Inc. Detect and recognize faces with Luxand FaceSDK. 2013. Online: http://luxand.com/facesdk/index_c.php. Last accessed: 2013-08-06.

[35] Mactaris. Webcam Settings 2.0 will support FaceTime HD camera on MacBook Air 2013. July 2013. Online: http://mactaris.blogspot.com/2013/07/webcam-settings-20-will-support.html.

[36] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp)iPhone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In George Danezis and Vitaly Shmatikov, editors, *Proceedings of CCS 2011*. ACM Press, October 2011. Online: http://www.cc.gatech.edu/~traynor/papers/traynor-ccs11.pdf.

[37] *1/6-Inch SOC VGA CMOS Digital Image Sensor: MT9V11212ASTC*. Micron Technology, Inc., 2005. Online: http://download.micron.com/pdf/datasheets/imaging/MT9V112.pdf.

[38] Charlie Miller. Owning the fanboys: Hacking Mac OS X. Presented at Black Hat Japan Briefings, October 2008. Online: https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Miller/BlackHat-Japan-08-Miller-Hacking-OSX.pdf.

[39] Charlie Miller. Battery firmware hacking: Inside the innards of a smart battery. Presented at Black Hat Briefings, August 2011. Online: http://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf.

[40] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. TapPrints: Your finger taps have fingerprints. In Srinivasan Seshan and Lin Zhong, editors, *Proceedings of MobiSys 2012*. ACM Press, June 2012. Online: http://synrg.ee.duke.edu/papers/tapprints-final.pdf.

[41] HD Moore. A penetration tester's guide to IPMI and BMCs. July 2013. Online: https://community.rapid7.com/community/metasploit/blog/2013/07/02/a-penetration-testers-guide-to-ipmi.

[42] Martha T. Moore. Pa. school district's webcam surveillance focus of suit. *USA Today*, May 2010. Online: http://usatoday30.usatoday.com/tech/news/surveillance/2010-05-02-school-spy_N.htm.

[43] Andy O'Donnell. How to secure your webcam in less than 2 seconds. March 2011. Online: http://netsecurity.about.com/b/2011/03/25/how-to-secure-your-webcam-in-less-than-2-seconds.htm.

[44] Devon O'Neil. Reflecting on Tony Hawk's 900. *ESPN*, May 2014. Online: http://xgames.espn.go.com/events/2014/austin/article/10622648/twenty-years-20-firsts-tony-hawk-900.

[45] *Oracle VM VirtualBox®*. Oracle Corporation, 2013. Online: http://www.virtualbox.org/manual/.

[46] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. ACCessory: Password inference using accelerometers on smartphones. In Rajesh Krishna Balan, editor, *Proceedings of HotMobile 2012*. ACM Press, February 2012. Online: http://www.hotmobile.org/2012/papers/HotMobile12-final42.pdf.

[47] Phisher Cat. Webcam light scaring slaves.... July 2013. Online: http://www.hackforums.net/showthread.php?tid=3660650. Registration required. Last accessed 2013-08-06.

[48] Jason Pisani, Paul Carugati, and Richard Rushing. USB-HID hacker interface design. Presented at BlackHat Briefings, July 2010. Online: https://media.blackhat.com/bh-us-10/presentations/Rushing/BlackHat-USA-2010-Rushing-USB-HID-slides.pdf.

[49] Sam Prell. Watch the Spelunky run that set a new high score record. *joystiq*, February 2014. Online: http://www.joystiq.com/2014/02/23/watch-the-spelunky-run-that-set-a-new-high-score-record/.

[50] Reinhard Riedmüller, Mark M. Seeger, Harald Baier, Christoph Busch, and Stephen D. Wolthusen. Constraints on autonomous use of standard GPU components for asynchronous observations and intrustion detection. In Anna Brunstrom and Svein J. Knapskog, editors, *Proceedings of IWSCN 2010*. IEEE Computer Society, May 2010. Online: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5497999.

[51] Thomas Ristenpart, Gabriel Maganis, Arvind Krishnamurthy, and Tadayoshi Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. In Paul Van Oorschot, editor, *Proceedings of USENIX Security 2008*. USENIX, July 2008. Online: http://adeona.cs.washington.edu/papers/adeona-usenixsecurity08.pdf.

[52] Thomas Ristenpart, Gabriel Maganis, Arvind Krishnamurthy, and Tadayoshi Kohno. Adeona. October 2009. Online: http://adeona.cs.washington.edu/index.html.

[53] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. Fast hash-based signatures on constrained devices. In Gilles Grimaud and Françis Xavier Standaert, editors, *Proceedings of CARDIS 2008*. Springer, September 2008. Online: https://www.hgi.rub.de/hgi/publikationen/fast-hash-based-signatures-constrained-devices/.

[54] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In Adrian Perrig, editor, *Proceedings of NDSS 2011*. Internet Society, February 2011. Online: http://www.cs.indiana.edu/~kapadia/papers/soundcomber-ndss11.pdf.

[55] Stop Police Brutality. Cop breaks 84 year old's neck for touching him. December 2013. Online:

http://www.policebrutality.info/2013/12/cop-breaks-84-year-olds-neck-for-touching-him.html.

[56] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In Peng Ning, editor, *Proceedings of NDSS 2013*. Internet Society, February 2013. Online: http://www.internetsociety.org/sites/default/files/02_2_0.pdf.

[57] Alexander Tereshkin and Rafal Wojtczuk. Introducing ring −3 rootkits. Presented at Black Hat Briefings, July 2009. Online: http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf.

[58] The iPatch. The iPatch. 2011. Online: http://www.theipatch.com. Last accessed: 2013-08-07.

[59] Craig Timberg and Ellen Nakashima. FBI's search for 'Mo,' suspect in bomb threats, highlights use of malware for surveillance. *The Washington Post*, December 2013. Online: http://www.washingtonpost.com/business/technology/fbis-search-for-mo-suspect-in-bomb-threats-highlights-use-of-malware-for-surveillance/2013/12/06/352ba174-5397-11e3-9e2c-e1d01116fd98_story.html.

[60] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. GPU-assisted malware. In Jean-Yves Marion, Noam Rathaus, and Cliff Zhou, editors, *Proceedings of MALWARE 2010*, pages 1–6. IEEE Computer Society, October 2010. Online: http://dcs.ics.forth.gr/Activities/papers/gpumalware.malware10.pdf.

[61] Vimicro. *VC0358 USB 2.0 Camera Processor*, September 2012. Online: http://www.vimicro.com/english/product/pdf/Vimicro_VC0358_PB_V1.1.pdf.

[62] Zhaohui Wang and Angelos Stavrou. Exploiting smart-phone USB connectivity for fun and profit. In Patrick Traynor and Kevin Butler, editors, *Proceedings of ACSAC 2010*, pages 357–366. ACM Press, December 2010. Online: http://dl.acm.org/citation.cfm?id=1920314.

[63] Richard Wheatsone. Video: CCTV footage shows the moment masked armed robber holds gun at terrified shopkeeper's head. *Mirror*, February 2014. Online: http://www.mirror.co.uk/news/uk-news/urmston-armed-robbery-cctv-footage-3149475.

[64] ZTech. QuickCam Pro 9000 LED Control. Post on the Logitech Forums, February 2008. Online: http://forums.logitech.com/t5/Webcams/QuickCam-Pro-9000-LED-Control/td-p/186301. Last accessed: 2013-08-06.

A Virtual machine escape

The reprogramability of the iSight firmware can be exploited to effect a virtual machine escape whereby malware running in a guest operating system is able to escape the confines of the virtual machine and influence the host operating system.

One method is to reprogram the iSight from inside the virtual machine to act as a USB Human Interface Device (HID) such as a mouse or keyboard. Once the iSight reenumerates, it would send mouse movements and clicks or key presses which the host operating system would then interpret as actions from the user.

To demonstrate the feasibility of a virtual machine escape from a VirtualBox virtual machine, we implemented a USB HID keyboard which, once loaded, performs the following actions in order:

1. send a "host key" press;
2. send command-space to open *Spotlight*;
3. send the key presses for "Terminal.app" one at a time;
4. wait a few seconds, send a return key press, and wait a few more seconds for the *Terminal* to open;
5. send the key presses for a shell command followed by a return key press;
6. disconnect from the USB bus and modify its USB device descriptor to use the product ID `0x8300` — the PID for the iSight in its unprogrammed state; and
7. reenumerate.

The VirtualBox host key, which defaults to the left command key on a Mac host, releases keyboard ownership, causing the rest of the key presses to go to the host operating system rather than to the guest operating system [45, Chapter 1].

Figure 4 shows an iSight that has been reprogrammed from inside a VirtualBox virtual machine sending key presses to *Spotlight*, instructing it to open *Terminal.app*.

When a new keyboard is first plugged into the computer, the *Keyboard Setup Assistant* asks the user to press several keys in order to determine the keyboard layout. This behavior appears to be controlled by the vendor ID, product ID, and device release number. By using the appropriate values for an Apple USB keyboard, the assistant

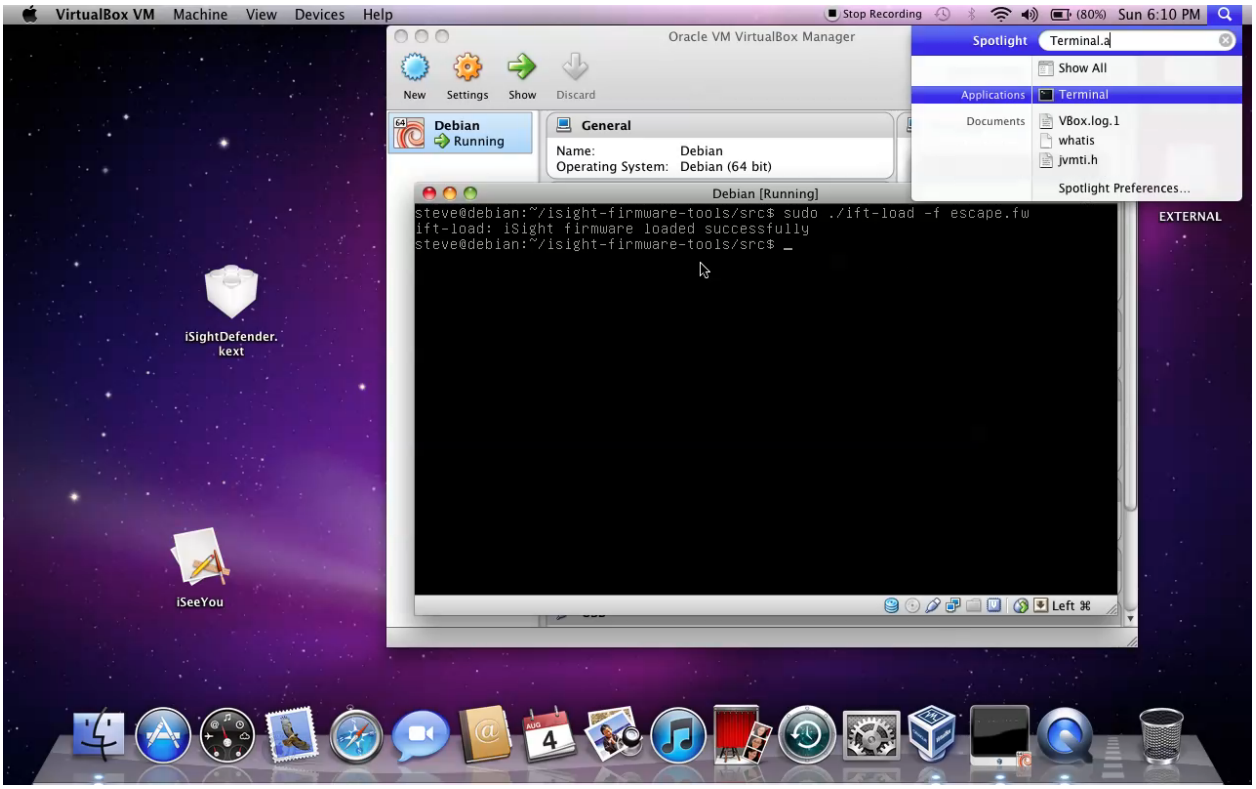


Figure 4: Virtual machine escape. The iSight has been reprogrammed to act as a USB keyboard from inside a VirtualBox virtual machine and it is sending key presses to the host operating system. It is in the middle of entering "Terminal.app" into *Spotlight*.

does not appear and there is no visual indication that the operating system believes a new keyboard has connected.

The shell command entered into the *Terminal* is unconstrained and could, for example, use *curl* to download arbitrary, new code and run it.

After the iSight has finished typing commands, it reenumerates as an unprogrammed iSight which causes the *AppleUSBVideoSupport* driver to reprogram it with the legitimate iSight firmware, removing evidence that the iSight was the infection vector.

Although we use the iSight to escape from the virtual machine, in theory, any EZ-USB device which is accessible from inside the virtual machine can be reprogrammed to behave as a HID keyboard.

The one major limitation is that the USB device must be connected to virtual machine before the attack is possible. By default, virtual machine monitors do *not* provide this connection for most devices and thus malware would need to coerce the user into establishing the connection.

Even with the device connected to the virtual machine, there is no feedback to the firmware that the attack is proceeding as planned. All it can do is send key presses in response the USB polling. If the user is sitting in front of the computer, the key presses sent by the iSight may be apparent and the user can interfere by performing an action such as typing or clicking the mouse. One way to partially compensate is to decrease the USB polling interval by changing the USB endpoint descriptors in the firmware allowing the iSight to send key presses more quickly.

Each operating system has its own policy which governs a process's ability to send USB device requests. On Linux, this is controlled by *udev*. In Figure 4, we used *sudo* inside the virtual machine to bypass the access controls of the guest operating system. Alternatively, the appropriate permissions could be granted to the user. One of these steps is required even though the host operating system, OS X, imposes no restrictions on the use of USB device requests. Since each guest operating system controls access to the USB device once it has been connected to the virtual machine, to perform an escape, malware must first acquire sufficient privileges in the guest operating system to reprogram the camera — a potentially nontrivial feat.

From the Aether to the Ethernet – Attacking the Internet using Broadcast Digital Television

Yossef Oren, Angelos D. Keromytis
Columbia University

25th May 2014

Abstract

In the attempt to bring modern broadband Internet features to traditional broadcast television, the Digital Video Broadcasting (DVB) consortium introduced a specification called Hybrid Broadcast-Broadband Television (HbbTV), which allows broadcast streams to include embedded HTML content which is rendered by the television. This system is already in very wide deployment in Europe, and has recently been adopted as part of the American digital television standard.

Our analyses of the specifications, and of real systems implementing them, show that the broadband and broadcast systems are combined insecurely. This enables a large-scale exploitation technique with a localized geographical footprint based on radio frequency (RF) injection, which requires a minimal budget and infrastructure and is remarkably difficult to detect. Despite our responsible disclosure to the standards body, our attack was viewed as too expensive and with limited pay-off to the attackers.

In this paper, we present the attack methodology and a number of follow-on exploitation techniques that provide significant flexibility to attackers. Furthermore, we demonstrate that the technical complexity and required budget are low, making this attack practical and realistic, especially in areas with high population density – in a dense urban area, an attacker with a budget of about $450 can target more than 20,000 devices in a single attack. A unique aspect of this attack is that, in contrast to most Internet of Things/Cyber-Physical System threat scenarios where the attack comes from the data network side and affects the physical world, our attack uses the physical broadcast network to attack the data network.

1 Introduction

The battle for the living room is in full swing. After being used for decades as purely passive terminals, our television sets have become the subject of intense, competitive attention. Technology companies wish to use the Internet to create a viewing experience which is more engaging, interactive, and personalized, and in turn maximize their ad revenue by offering advertising content which is better targeted at the user. As the result of this trend, most US and European households with broadband Internet access now have at least one television set which is also connected to the Internet [37, 27], either directly or through a set-top box or console. In technical terms, a device which has both a **broadcast** TV connection and a **broadband** Internet connection is called a **hybrid terminal**. The specification that defines how to create and interact with "hybrid content" (which combines both broadcast and broadband elements) is called **Hybrid Broadcast-Broadband Television**, or *HbbTV*.

At its core, HbbTV combines broadcast streams with web technologies. The broadcast channel, augmented with the notion of separate digital streams, allows the transmission of distinct yet intertwined types of content that enable rich-interaction experience to the user. However, this enhanced interaction introduces new vulnerabilities to what was until now a conceptually simple network (TV broadcasting) and media-presentation device.

This paper examines the security impact of emergent

properties at the intersection of digital video broadcasting and web technologies. The work presented here is based both on analysis of the HbbTV standard and on experimentation with actual DVB hardware. The attacks were crafted using low-cost hardware devices using open-source software, and they are extremely easy to replicate.

While the impact of many of these attacks is exacerbated by poor implementation choices, for most attacks the core of the problem lies with the overall architecture, as defined in the specification itself. Thus, our findings are significantly broader than the specific devices that we used in our analysis; indeed, *any* future device that follows these specifications will contain these same vulnerabilities. Exploiting these vulnerabilities, an attacker can cause many thousands of devices to interact with any website, even using any credentials stored in the TV sets for accessing services such as social networks, webmail, or even e-commerce sites. This capability can be leveraged to perform "traditional" attack activities: perform click-fraud, insert comment or voting spam, conduct reconnaissance (within each home network or against a remote target), launch local or remote denial of service attacks, and compromise other devices within the home network or even elsewhere. Beyond these, the attacker can also control the content displayed on the TV, to craft phishing and other social engineering attacks that would be extremely convincing, especially for TV viewers who are educated to (and have no reason not to) trust their screens. Finally, the attacker can use the broadcast medium to effectively distribute exploits that completely take over the TV set's hardware. Most of these attacks require no user knowledge or consent – the victims are only required to keep watching their televisions. The unique physical characteristics of the broadcast TV medium allow these attacks to be easily amplified to target tens of thousands of users, while remaining **completely undetectable**. Remarkably, the attacker does not even require a source IP address.

Today's smart TVs are already very complex devices which include multiple sensors such as cameras and microphones and store considerable amounts of personal data. Equipment manufacturers are busy adding more hardware and software capabilities to these devices, with the aim of turning them into the center of the user's digital life. Obviously, as smart TVs become more capable, and as users use them for more sensitive applications, the impact of the attacks described here will only grow.

One interesting, perhaps unique aspect of the problem space we are examining here is the reversal of attack source and destination domains: in typical attacks against Internet-connected physical systems, large-scale device compromise through the data network can lead to physical exploitation with a large (perhaps global) geographical footprint. With HbbTV, a physical attack with a relatively large geographical footprint can lead to large-scale data network compromise, at least in areas with high population density. The essence of the problem we address lies in that the hybrid TV now connects the broadcast domain, which has no authentication or protection infrastructure, to the broadband Internet domain. This allows the attacker to craft a set of attacks which uniquely do not **attack the TV** itself, but instead **attack through the TV**.

1.1 Disclosure and response

Our work addresses a security risk in a specification which is already in very wide use in Europe, and is on the verge of expanding to the US and to the rest of the world. We thus made an effort to responsibly disclose our work to the relevant standards bodies. In December 2013, we provided a description of our RF-based attack, together with a video recording of an attack in progress, to the HbbTV Technical Group. In January 2014 we were informed that the HbbTV Technical Group discussed our disclosure, but did not consider the impact or severity of these attacks sufficient to merit changes to the standard. There were two main criticisms raised by the HbbTV Technical Group. The first criticism was that it would be very difficult for the attacker to reach a large number of systems; the second was that, even when an attack is carried out, a Smart TV has a very limited attack surface, so attacks would not be cost-effective. We explicitly structured this paper to address both of these criticisms – we quantitively demonstrate how a low cost attack can reach thousands of systems, and we show how attacks can cause a considerable amount of damage and provide a real financial gain for the attacker.

Document Structure: The rest of the document is arranged in the following manner: Section 2 provides a high-level overview of digital video broadcasting. In Section 3, we describe the fundamental weaknesses of the protocol which enable our attack and propose an attack

setup designed to exploit them. Next, in Section 4 we describe a series of possible attacks based on these weaknesses. We continue in Section 5, where we quantitatively analyze the impact potential of our attack, based on the power and propagation characteristics of the attack setup and on actual demographic information. In Section 6 we experimentally verify several of our proposed attacks. In Section 7 we analyze the financial impact our attacks and evaluate several possible countermeasures. Finally, we conclude in Section 8.

2 Fundamentals

The vision of an Internet-powered living room brings to mind products such as on-demand video streaming or cloud-delivered gaming. However, the masters of the living room are still the incumbent operators of existing television broadcast networks, who broadcast their content to billions of viewers worldwide. In order to compete with the new generation of entertainment content, the operators of these broadcast networks are also looking for ways to add Internet-based functionality to their traditional content. For example, a broadcast television channel might use Internet functionality to ask its viewers to participate in an online poll, or to vote for a candidate in a game show. The broadcast channel might also invite the viewer to learn more about an advertised product using interactive web content, or even replace regular broadcast advertisements with custom-delivered Internet ads personalized to the particular user. In this form of content delivery enhances traditional broadcast content with an interactive HTML overlay, rendered by the TV together with the normal broadcasted channel. This content is commonly called "**Red Button Content**", since pressing the red button on the TV remote is (by convention) the standard way of interacting with it.

The specification defining this behavior is called **Hybrid Broadcast-Broadband Television**, or HbbTV, and it is maintained by the European standards body ETSI [10]. The current generation of the specification, version *1.2.1*, is enjoying very rapid adoption and is in active deployment or in advanced stages of testing in most of Europe. In December 2013, the Advanced Television Systems Committee (ATSC), which defines the digital video standards in the US, Canada, South Korea and several other countries, published a candidate standard for hybrid TV in America [6]. This candidate standard shares much of its structure with the European HbbTV standard, and is specifically equivalent to the European standard with respect to the attacks described in this paper.

HbbTV is designed to work on top of a standard Digital Video Broadcasting (DVB) system. While DVB can be delivered over cable, satellite or standard terrestrial signal, each with its unique radio frequency (RF) modulation and transmission scheme, the underlying digital stream is essentially the same for all delivery methods. This stream takes the form of an MPEG-2 **Transport Stream** [23], which multiplexes together multiple data streams named MPEG-2 **Elementary Streams**. Each elementary stream carries an individual element of a television channel, such as video, audio or subtitles. Special **metadata streams,** which the specification refers to as **information tables,** are then used to group together multiple elementary streams into an individual TV channel and provide additional information about the channel such as its name, its language and the list of current and upcoming programs. A single radio physical frequency may thus carry multiple channels.

2.1 Mixing broadcast and broadband

The HbbTV specification extends standard DVB by introducing additional metadata formats which mix broadband Internet content into the digital television channel. While the specification proposes multiple ways in which web content can be used in a TV, this article will focus on the most common form of content, **autostart broadcast-dependent applications**. This form of content starts running automatically when the user tunes into a particular TV channel, and terminates when the user moves to another channel. To create an autostart broadcast-dependent application, the broadcaster includes in the MPEG transport stream an additional **application information table** (AIT) describing the broadband-based application, then references this table in the **program mapping table** (PMT) describing a certain TV channel. The HbbTV specification defines two possible ways of providing the application's actual web content (*i.e.,* HTML pages, images, and scripts). One way is to have the AIT include a URL that points to a web server hosting the application. Another possible way is to create an additional data

stream which includes the HbbTV application's HTML files, deliver this additional elementary stream over the broadcast transport, and finally have the AIT point to this data stream. The way in which the latter embedding method was realized leads to a serious security problem, as we discuss later.

Regardless of the delivery method, Internet content is rendered by the TV using a specially-enhanced web runtime, described in the HbbTV standard as a Data Execution Environment (DAE) [14]. In addition to the document object model (DOM) elements available to normal HTML environments such as XmlHttpRequest, the DAE exposes additional DOM elements which are specific to the television world (for example, information about the running program and the current channel). The DAE also allows programatic access to the live TV broadcast window. Thus, it is possible for an HbbTV application to render content on top of the TV broadcast, resize the broadcast window or even completely replace the broadcasted content with its own content. On the other extreme, it is also possible for an HbbTV application to run without displaying any indication to the user. Practically speaking, most "benign" applications typically display a small overlay inviting the user to press the Red Button, then disappear to run transparently in the background.

2.2 Security in HbbTV

Smart TVs are built with some consideration of security, since they are often used to display content protected by digital rights management (DRM) schemes. Indeed, the HbbTV specification dedicates an entire chapter to security, but the discussion is mainly focused on protecting DRM content and not on other aspects of security. To that effect, the HbbTV specification describes trusted and untrusted applications, and restricts "sensitive functions of the terminal" only to trusted applications. Examples of such "sensitive functions" include downloading and playing back DRM-protected downloaded content (actions which may incur a cost on the viewer), as well as configuring and activating the terminal's scheduled recording (time-shifting) capabilities.

The attacks described in this work make use of capabilities which are available both to trusted and untrusted applications. None of the attacks described in this work are restricted in any way by HbbTV's security mechanisms. Furthermore, since the specification does not strictly define how an application can become trusted, it might be possible to inject an attack into a trusted application without changing its trusted status.

3 Attack Characterization

Several unique properties of HbbTV make it potentially prone to attack. These security weaknesses can all be considered **emergent** properties, which exist on the boundary between the broadband and broadcast systems, and stem from the different expectations and guarantees which exist in each system.

First and foremost, HbbTV applies a very problematic security model to web content embedded into the broadcast data stream. This is, in our opinion, the most serious security flaw in HbbTV, and one which has not been discussed in any previous work. One of the cornerstones of modern web security is the Same-Origin Policy [1], which essentially serves to isolate content retrieved from different origins and prevent content from one web site from interfering with the operation of another web site. Under the same origin policy, each piece of web content is provided with an origin consisting of a tuple of scheme, host and port, and two resources are limited in their communications unless they share the same origin.

When an HbbTV application is downloaded from the Internet via URL, the origin of the web content is clearly defined by the URL, appropriately isolating HbbTV applications to their own domain and preventing them from interfering with Internet at large. However, when the content is embedded in the broadcast data stream it is not linked to any web server and, as such, has no implicitly defined origin. The HbbTV specification suggests [10, S 6.3] that in this case the broadcast stream should **explicitly define its own web origin** by setting the `simple_application_boundary_descriptor` property in the AIT to any desired domain name.

The security implications of this design decision are staggering. Allowing the broadcast provider control over the purported origin of the embedded web content effectively lets a malicious broadcaster inject **any script of his choice** into **any website of his choice**.

An illustrative example of such an attack is presented in Figure 1. In this attack, which we dis-

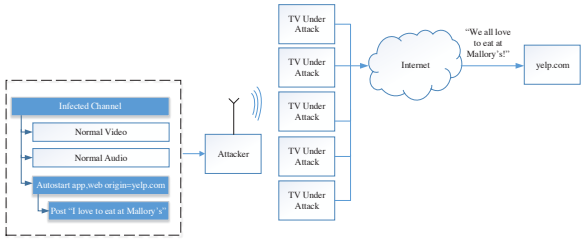


Figure 1: A practical attack based on a malicious HbbTV application. In this attack the malicious player forces multiple infected TVs to interact with a rating site and leave a favourable review for his restaurant.

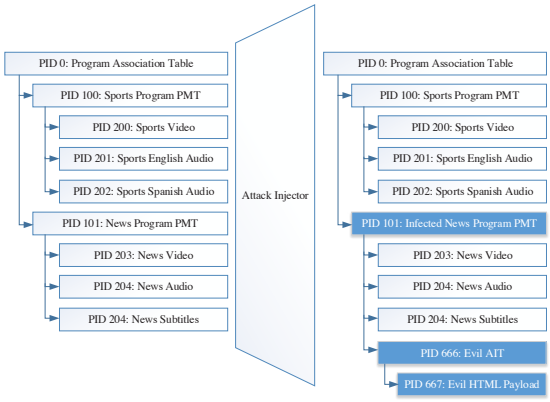


Figure 2: Injecting a malicious appication into a DVB stream. Note that only the program mapping table is modified, while the audio and video content is left untouched.

cuss more extensively in Subsection 4.2, the adversary delivers a malicious Javascript payload over HbbTV, and furthermore indicates by the `simple_application_boundary_descriptor` property in the AIT that this payload's web origin is a rating site. Next, the attacker has the TV render a simple HTML page which embeds the real rating site's home page (downloaded from the broadband Internet), as well as this script, in a zero-sized frame. The malicious script now has full programmatic access to the content delivered by the rating site, since it is running within the same web origin. To make matters worse, if the user has previously logged on to the site, this attack allows the attacker to fully interact with the website on the user's behalf. While the innocent viewers enjoy their normal television content, the malicious application causes their infected TVs to interact with the rating site over the Internet to leave favourable reviews for the attacker's restaurant or to harrass his competitors.

3.1 General Principle of Operation

We now describe how an attacker can use the vulnerability described above to launch a series of large-scale attacks. Our setup targets digital terrestrial television (DTT), which is the most common way in which television is received in many parts of the world [11]. In Subsection 7.2 we discuss how this attack can also be applied to other delivery methods such as cable or satellite.

Our attack works by creating a television broadcast which includes, together with the normal audio and video streams, a malicious HbbTV application. To maximize the effectiveness of our attack, we would like this as many users as possible to tune into this broadcast. The best way to do so is to carry out a form of **man-in-the-middle attack**, in which the attacker transparently modifies a popular TV channel to include a malicious payload.

Our attack module follows the general design illustrated in Figure 2. Following the notation of Subsection 2.1, the attacker adds into the intercepted stream a new **Application Information Table**, as well as a **data stream containing a malicious HbbTV application**, which the new AIT points to. The attacker then modifies one or more existing **Program Mapping Tables** to reference the new malicious application, while leaving the audio and video contents of the channel unmodified. It is important to note that the attacker does not have any form of control or cooperation with the radio tower itself.

The physical attack setup required by the attacker is illustrated in Figure 3. The attacker's uses a **receive antenna** connected to a **DVB tuner** to intercept a legitimate television signal, **modifies the content** of the DVB stream to add its malicious payload, and finally uses a **DVB modulator** connected through a **power amplifier** to a **transmit antenna** to re-transmit the modified signal to the **TV under attack** using the same frequency as the original broadcast. The TV under attack is, in turn, connected to the Internet.

Our attack works because in a certain geographic area

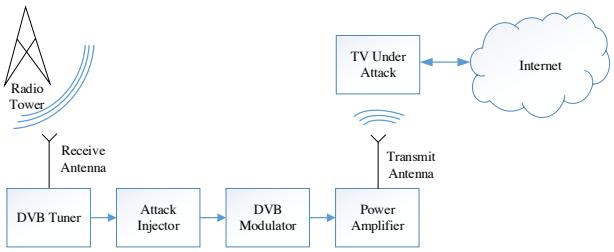


Figure 3: Attack Setup

around the attacker the malicious modified signal will be stronger than the original signal transmitted by the tower. This will cause any televisions in the area to immediately fall victim to the attacks described below. We note that since in digital broadcasting multiple TV channels are sent from the radio tower using the same radio frequency, a single attack setup is capable of injecting attack code into several channels simultaneously.

The characteristics and estimated cost of each of the components in Figure 3 are presented below:

Receive antenna and DVB tuner – a USB-powered DVB tuner and a short passive antenna can be purchased online for about $15. The open-source VLC media player [33] is capable of interfacing with many of these tuners and sending the demodulated stream extracted from an entire RF channel to a file or a network socket.

Content modification – the demodulated stream is modified to contain a malicious application (either as a URL, or as a full application delivered via data stream), and the PMTs of all TV channels in the demodulated stream are modified to auto-start this application as soon as the user tunes into the channel. Since the video and audio streams in the channel are forwarded without any modification, this operation is not particularly computation intensive, and any low-cost computer with USB 2.0 support can be used for this purpose. A software suite named Avalpa OpenCaster [8] provides a set of open-source command-line tools which can be used to modify a multiplexed DVB stream in real time.

DVB modulator – this hardware component takes a multiplexed MPEG stream and converts it into an RF signal suitable for transmission. While these devices were once massive and expensive, modern DVB modulators are remarkably small and easy to use – a full-featured USB-powered modulator which can interface with OpenCaster can be purchased online for less than $200.

Power amplifier and transmit antenna – the attacker needs to create a signal that is stronger than the original TV tower's signal and transmit it toward the target televisions. An attacker with a higher transmit power can affect more television sets, but a high-power setup is generally less portable, giving the attacker a higher probability of being detected and arrested. In Section 5 we formally analyze the power requirements of the attacker and show that, under the right conditions, a remarkably high amount of television sets can be affected with a moderate-to-low powered amplifier.

3.2 Additional Security Weaknesses

3.2.1 Attacks are untraceable

In traditional Internet-borne attacks, it is always assumed that the attacker is himself present on the Internet before he can deliver a malicious payload to his victims. The attacker's IP and DNS entries can then be used by intrusion protection services and law enforcement agencies to protect against the attack as it occurs, and to trace and prosecute its perpetrators after it has concluded. In contrast, our attacker needs no such infrastructure to deliver its malicious payload. It is surprisingly simple and inexpensive to build a digital terrestrial television (DTT) transmitter and use it to reach thousands of potential hosts. After the attack concludes, the attacker leaves no trace of his activities in the form of IP or DNS transactions.

Operating an unlicensed TV transmitter is illegal in many countries. Law enforcement agencies capture these illegal transmitters by **triangulation** methods, which involve sending multiple car-mounted receivers to the vicinity of the attack, then using the differences in received signal strengths between receivers to locate the rogue transmitter. A sensitive receiver can also "fingerprint" the rogue transmitter's RF envelope and help recognize it in the future. While this defense mechanism can potentially trace our radio attacker, mobile triangulation is a **reactive** defense step, which requires a considerable expense of time and resources from the defender's side. Considering that the attack we describe has a very limited geographical signature, operates for a very limited time (potentially only a few minutes), and causes no visible adverse effects

to the user, it is highly unlikely that the attacker will be caught by these methods.

3.2.2 Attacks are invisible and unstoppable

HbbTV content is not required by standard or convention to offer any visual indication that it is running. Depending on the choice of the application creator, HbbTV content can run invisibly in the background, side by side with the broadcast content, or even take over part or all the user's entire screen. At one extreme, this makes it possible for HbbTV applications to run completely in the background without the knowledge or consent of the user. In [21] Herfurt discovered that many German broadcasters are using this functionality of HbbTV to invisibly track the viewing habits of users by periodically "phoning home" while the TV is tuned to a particular channel. At the other extreme, it is possible for an HbbTV application to take over part or all of the user's screen without his knowledge. Herfurt used this functionality to demonstrate a proof-of-concept application that replaces the news ticker of a German news channel with headlines from a satire website.

Another related weakness is the weak control the user has over the life-cycle of HbbTV applications. As described in Subsection 2.1, an application can start running automatically as soon as the user starts viewing a certain channel. More troubling is the fact that, once an HbbTV application has started running, there is no standard way of **stopping** it, short of switching a channel, turning off the television, or completely disabling HbbTV support.

4 Attacks

The attacks proposed in this Section are based upon our analysis of the HbbTV standards, as well as upon personal communications with the HbbTV technical group, who have confirmed that our attacks are possible given the current specification. Some of these attacks described below can be applied even to perfectly secure Smart TV implementations with no known exploits; Other attacks allow the attacker to transform local vulnerabilities on the Smart TV into automatic, large-scale distributed exploits. With the exception of the attack described in Subsection 4.5, all of these attacks take place without the user's knowledge or consent, requiring the user to do nothing more than keep his TV turned on and tuned to his favourite channel.

4.1 Distributed Denial of Service

To carry out this attack, the attacker creates a simple Red Button application which repeatedly accesses a target website with high frequency, using a simple mechanism such as a zero-sized `iframe` element or through repetitive calls to `XmlHttpRequest`. All TVs tuned to the infected channel will immediately start running the application, potentially overwhelming the target website. Due to the design of the HbbTV specification, the owners of TVs who are carrying out this attack have no knowledge that they are participating in this attack, nor do they have any way of stopping it.

This attack is the simplest abuse of the HbbTV protocol, and was also considered by [21], albeit in a different attacker model. As scary as this attack sounds, we note in Subsection 7.1 that there are far less expensive and risky ways of DDoSing a website.

4.2 Unauthenticated Request Forgery

This attack is similar to the previous attack, but this time the infected users do not blindly access the site under attack, but instead attempt to interact with it in a meaningful manner. For example, such an attack could skew the results of an online poll or competition, "spam" a forum with comments to the point of unreadability, falsely promote another website by "liking" or "upvoting" it, or falsely obtain ad revenue by programmatically clicking on an ad (a.k.a. "click fraud"). This attack venue is especially painful for the designers of HbbTV, since the entire point of the specification is to allow this type of interaction between TV viewers and websites.

This attack is a variant of traditional cross-site request forgery (CSRF) attacks, which are well-known to the security community [2]. However, one unique advantage of the HbbTV attack vector is that the attack is not "blind" – due to the unique way same-origin is implemented for HbbTV, the attack script can fully interact with the static and dynamic content of the page with the full permissions of a human user accessing the webpage. This defeats many of the state-of-the-art defenses against CSRF, which operate by embedding session and authentication

tokens in locations which are only accessible within the same origin as the protected web page.

4.3 Authenticated Request Forgery

An interesting twist on the previous attack, this attack assumes that the user has previously authenticated to a certain website using another application on his Smart TV, and that the TV now holds a cookie, an HTML5 local storage element, or any other authentication token for this website[1]. When the infected application accesses the website, it will now automatically do so with the full credentials of the logged-in user, a fact which dramatically increases the damage potential of the previous attack. An infected application using this attack vector can, for example, post links to malware to the legitimate user's friends over Twitter or Facebook, purchase DRM-protected content whose royalties are pocketed by the attacker, or call a premium number using a VoIP application. As the usage scenarios of Smart TVs grow and users begin using them for more applications such as e-commerce and health, the damage potential of this attack will grow rapidly.

4.4 Intranet Request Forgery

This attack makes use of the fact that the Smart TV is most likely connected to a home wireless network shared with other devices such as wireless routers, personal computers and printers. Instead of attacking the whole Internet, the attacker instead mounts his attacks on those local intranet devices. The most basic attack would be a port scan to discover which devices are present on the home network (this can assist in planning a burglary). If vulnerable devices are discovered on the network, the attacker can also try and exploit them using the Smart TV. For example, the attacker can identify a vulnerable wireless router and a Windows PC, then proceed to modify the DNS settings of the router so that the PC is directed to a phishing website when it attempts to connect to a banking website. This attack, which again has been investigated in other works such as [26], is particularly effective due to the way same-origin is implemented on HbbTV. Remarkably, the attacker's code can freely interact with the device under attack and observe the results of its interaction, without requiring additional steps such as DNS rebinding.

[1] While the smart TV platform we evaluated had two separate "web runtimes" – one for the TV and one for the HbbTV stack – and thus kept credentials isolated, this behavior was probably caused by engineering concerns (two independent teams may have written the two runtimes, with no time for integration) and is in no way required by the standard.

4.5 Phishing/Social Engineering

As described in Subsection 3.2.2, HbbTV content is displayed on the user's television without any warning or notification, and the user cannot turn it off without turning off the TV itself. HbbTV content can completely overlay the user's TV broadcast and can programmatically interact with many of the buttons on the user's remote control. This direction, also investigated by Herfurt in [21], makes HbbTV content a natural vector for attacks which mislead the user into divulging sensitive information or otherwise acting in a harmful manner.

For example, a malicious HbbTV payload can notify the user that he must enter his credit card information to view some restricted content, compel the user to change the configuration of their network in a form that compromises their security (for example, instruct the user to press the WPS button on their wireless router, thus allowing a malicious device to join the network), or even encourage "real world" risky behavior, such as notifying the user that a "cable technician" is due to visit their house at a certain time and date, or that the TV needs to be "recalled" and physically delivered to the attacker. This attack is different than the other attacks described in this paper since it requires user interaction and, as such, is more likely to be detected or simply ignored. Obviously, the damage potential of this attack will increase in the future as more users are trained to interact with their TVs for applications other than passive content consumption.

4.6 Exploit Distribution

A modern smart TV is essentially a personal computer with a very limited user interface, running a highly modified version of Linux or Android. Just like normal PCs, security exploits are occasionally discovered in Smart TVs – either in the vendor's proprietary software, or in the device's various open-source underlying components. Just like normal PCs, Smart TVs also have automatic software update mechanisms which are generally successful

in keeping the TVs running smoothly and securely. However, the vulnerability-to-patch cycle for these devices is typically much longer than that of a desktop operating system, due to the additional steps required by the equipment vendor to implement, test and deploy security updates for this nonstandard platform. Whenever an exploit is discovered for a Smart TV platform, the combination of HbbTV's invisibility and undetectability make it a remarkably efficient method of distributing this exploit and compromising the TVs.

Assume, for example, that a Smart TV uses an open-source image processing library as part of its code. Assume now that a patch is released to fix a vulnerability in the upstream version of this component. While the equipment vendor is busy porting, testing and deploying a patch specifically tailored for the smart TV, an attacker can immediately craft an exploit corresponding to this vulnerability, embed it in a malicious Red Button application, then immediately deploy it to thousands of Smart TVs.

5 Scale Considerations

As stated in Subsection 1.1, one of the main criticisms directed at our work was the claim that it is very difficult for the attacker to infect large numbers of television sets. This Section quantitatively demonstrates how a low-budget attacker can modify and then overwhelm a TV tower's transmissions in a limited geographical area.

We first determine the approximate area an attacker can cover for a fixed transmit power. We assume that the **attack frequency** is approximately 500MHz, corresponding to the DVB-T UHF band. We assume that the unamplified signal exiting the attacker's modulator has a signal level of 0 dBm, and that the attacker uses an **omnidirectional antenna** which is in free space and on a level plane with the targeted devices. Thus, the attacker's output power is equal to the gain of his power amplifier G.

We further assume that the radio tower's original transmission is received by all targeted devices with a signal level T of -50 dBm, corresponding to a moderate to high signal level (digital television receivers can function at signal levels as low as -112 dBm, while the FCC defines the "City Grade" signal level for digital television at -61 dBm [5]).

Our final assumption is that, when receiving two competing DVB-T signals with the same frequency (a condition technically referred to as co-channel interference), the receiver will demodulate and display the stronger signal while ignoring the weaker signal. This assumption, which does not hold for analog transmission systems, is valid for DVB signals as long as the stronger signal overwhelms the weaker signal by some minimal amount (the International Telecommunications Union recommends in [24, Table 15] a power difference of 6 to 8 dB between the stronger station and the interfering weaker station, but a practical attacker who is not neccessarily standards compliant can get away with a much smaller margin).

The decay in decibels of a radio signal with frequency f (in Hz) over a distance d (in meters) is described by the Free Space Path Loss equation [30]:

$$FSPL = 20\log_{10}(d) + 20\log_{10}(f) - 147.55$$

We require that the attacker's signal will be more powerful than the radio tower's original signal:

$$G - FSPL > T$$

Assigning values to f and T we obtain that for a successful attack

$$G - 20\log_{10}(d) - 20\log_{10}\left(5 \cdot 10^8\right) + 147.55 > -50$$

Solving for d we obtain that

$$d < 10^{\frac{G}{20}+1.18}$$

Using this formula shows that with a 1 W (30 dBm) amplifier, whose cost is approximately $250, the attacker will be able to cover a region with radius of 477 m, or an area of 1.4 km^2. With a more powerful 25 W (44 dBm) amplifier, whose cost is approximately $1500, the attack can cover a region with radius 2385 m, or an area of 35 km^2. The attacker might have an incentive to use a lower-powered amplifier to reduce his risk of being detected by mobile triangulation methods (see Subsection 3.2.1).

Our next step was to demonstrate quantitatively that there exist densely-populated urban areas in which popular digital TV stations are received with a sufficiently low power level as to allow such an attack to be carried out.

Our analysis was based on the NASA SEDAC Metropolitan Statistical Areas dataset [35], which records

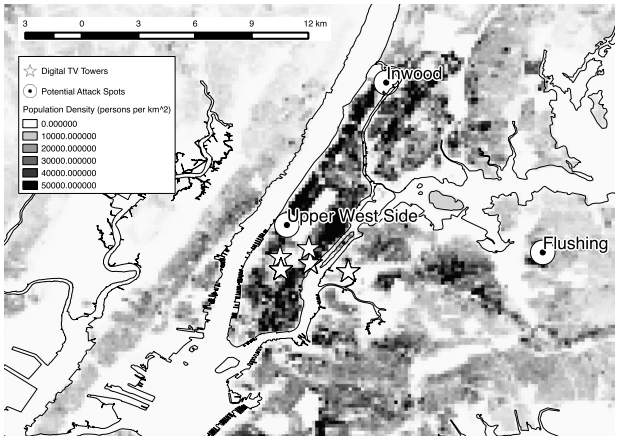


Figure 4: Attack potential – New York City. White stars identify digital TV towers. Bulls-eyes identify locations where an attack would be particularly effective.

demographic and socioeconomic data for 50 US cities, with a spatial resolution of approximately 250 square meters. We cross-correlated this dataset with the FCC database of digital TV towers in the United States and with station coverage maps supplied by TV Fool [13]. The TV Fool maps use 3D propagation modeling algorithms, and consider transmitter power, terrain obstructions and Earth curvature. Our objective was to find densely-populated areas in which a popular channel is received with a power of between -50 and -61 dBm. As a case study, we present our results for New York City in Figure 4 (more maps will be available in the extended version of this paper). The map shows the high-resolution population density of New York City, with the location of radio towers marked with white stars, and potential attack spots marked with bulls-eyes. Flushing, Queens, with a population density of 71,000 persons per km^2, is one example. There are 7 DTV stations with the desired characteristics in this location, most notably the Home Shopping Network. HSN can also be attacked in the Upper West Side, with a population density of over 80,000 persons per km^2. In certain locations in the Inwood area, where the population density is 50,000 persons per km^2, the attacker can infect 10 different stations, including CBS, NBC, Fox and the Spanish language Telemundo.

There are several limitations to this attack. The attacker obviously has to be physically present at the attacked location, and to have a line of sight both to the transmitter tower and to the antennas of the televisions under attack. In an urban setting this condition can be realised if the attack is carried out from the roof of an appropriately located tall building. To reduce the attacker's risk of capture and thus increase the effectiveness of the attack, the attacker can install the relay equipment on a remote controlled-drone and fly it to an appropriate location, similar to the work of [34].

The attacker would also need to prevent his receive antenna from picking up his own signal. This can be achieved by using a directional receive antenna directed toward the radio tower, together with a directional transmit antenna directed toward the TVs under attack, and finally locating the receiver setup in one of the transmitter's "dead zones". Using a directional antenna setup will change the shape, but not the general area, of the location under attack.

6 Experimental Validation

To show the validity of our claims, we created a test setup and experimentally reproduced a few of the attacks proposed in this paper. Our attacks were carried out on a modern Smart TV, manufactured in 2012 and running the latest software version supplied by the vendor. Our DVB demodulator was an OEM DVB-T stick based around the highly popular Afatech AF9015 chipset. The broadcast DVB stream was captured using VLC Player [33] running on Linux. Our DVB modulator was a DekTec DTU-215 unit, which was connected via USB to a low-cost laptop computer running Linux. For safety reasons our test setup did not include a power amplifier and transmitter antenna – instead, the DVB modulator was directly connected to the TV's antenna input through a 10 dB RF attenuator. The signal sent to the TV included different malicious HbbTV payloads created using the open-source OpenCaster package [8], version 3.2.1, and were played back to the TV using the DekTec StreamXpress software utility.

Using our test setup, we were able to create HbbTV applications which ran invisibly in the background, as well as applications which completely took over the TV screen. Using HbbTV, we were able to deploy the

Browser Exploitation Framework (BeEF) Toolkit [7] on the TV and use it to port scan the TV's intranet, examine the TV's runtime environment and display fraudulent login messages on the TV. We were able to crash the TV by having it render a malformed image file – a precursor for exploit distribution. Finally, we were able to craft a denial of service attack on an external web server, which ran as long as the user was tuned in to a particular channel. We verified that we were able to access servers both on the Internet at large and on the local intranet.

7 Discussion

7.1 Risk Assessment Analysis

Table 1 summarizes the attacks described in this paper and assigns each one with a qualitative complexity and damage potential. The justification for each qualitative complexity and damage assessment grade is provided below. In our analysis we assume the attack setup costs $450 in fixed costs, and that each attack costs an additional $50 per hour in variable costs (including equipment running costs and compensation for the risk taken by the attacker, who has to be physically close to the attacked location). We conservatively assume that the attack impacts 10,000 hosts – as we showed in Subsection 3.1, the attack can be easily scaled by one or more orders of magnitude by using a higher-powered amplifier.

The **denial of service** attack is the attack with the lowest complexity, since it requires no research on the side of the attacker, neither of the TV nor of the site under attack. However, its damage potential is also low, especially since it is not cost effective. As anecdotally shown in [3], a DDoS attack involving more than 20,000 hosts costs approximately $5 per hour. However, it must be noted that since the TV-based DDoS attack described here is localized to a single area, it can overwhelm a single edge node on a Content Distribution Network and thus deny service to other users in the same physical area.

The **unauthenticated request forgery** attack (in which an attacker uses HbbTV to vote in a poll, promote an article, or click an advertisement) also has low complexity, since it only requires minimal reverse engineering of the target web page. However, it has a higher damage potential than the DDoS attack, since it is much easier to monetise due to the possibility of click fraud [19]. According to Google's official figures, the average cost per click to advertisers in 2013 was $0.94, out of which 25% goes to the fraudulent advertiser [22]. This means the attacker can expect an income of around $2500 per attack even if every compromised host clicks only a single ad. In addition, since the interactive abilities abused by this attack are the main selling points of HbbTV, this attack has a wider area chilling effect of scaring advertisers and limiting the adoption of HbbTV.

The **authenticated request forgery** attack has a higher complexity than the previous two attacks, since it requires the attacker to discover and exploit a situation in which credentials are shared between the HbbTV runtime and other applications running on the Smart TV. However, this attack has a higher damage potential, since webmail and social network accounts are easier to monetise – according to [38] a verified Facebook account can retail for as much as $1.50, giving the attacker a potential income of $15,000 per attack. Once users begin using their Smart TVs for additional activities such as shopping the impact of this attack will only grow.

The **intranet request forgery** attack has medium complexity, since it involves compromising and exploiting not only the TV but also an intranet-connected device such as a router or a printer. However, there are existing intranet attacks which can be reused for this purpose. The damage potential of this attack is understandably high, since it lets the attacker compromise the user's personal computer.

The **phishing/social engineering** attack may be technically easy to launch, but it has external factors which make it more complex to carry out. First, the user's cooperation is required for this attack to succeed, raising the chance that the attack is ignored or, in the worst case, reported to law enforcement. In addition, the attack requires the attacker to set up additional attack infrastructure (e.g. a web server for collecting credentials), raising the risk of capture. The damage potential of this attack, however, is the highest of all attacks described here, since it risks the user's personal safety.

The **exploit distribution attack** may appear to be technically the most complex attack described here. However, since Smart TVs are commonly built using open-source components, an aspiring attacker can use an exploit patched in the most recent version of the component and not yet updated in the Smart TV. This attack has a

Attack Type	Complexity	Damage Potential	Overall Risk
Denial of Service	Low	Low	Medium
Unauthenticated Request Forgery	Low	Medium	High
Authenticated Request Forgery	Medium	High	High
Intranet Request Forgery	Medium	High	High
Phishing/Social Engineering	High	High	Medium
Exploit Distribution	Medium	High	High

Table 1: Risk assessment matrix of suggested attacks

high damage potential, since it results in total compromise of the TV.

7.2 Attacking cable and satellite

The physical attack setup described in the previous sections assumed a digital terrestrial television (DTT) broadcast system. According to [11], this is the most common delivery method for digital television across Europe. However, there are several areas of the world, most notably the USA, where this form of delivery is less common than cable or satellite communications. While splicing into cable connections or hijacking satellite signals is more expensive and risky than transmitting a low-power UHF signal, it might still be possible to attack such systems if they use microwave RF links for part of their (non-broadcast) transmission networks. Setting up a relay system which adds malicious applications to such a relay link is possible using fundamentally the same technique as the one described in this paper, with the limitation that the relay device must be physically located along the line of sight of the microwave link. High-budget adversaries such as crime syndicates or state players might also like to directly attack cable or satellite distribution centers to launch truly massive large scale attacks using HbbTV, gaining control over hundreds of millions of connected devices.

7.3 Countermeasures

As stated in Section 3.2, there are three main security weaknesses in HbbTV: the fact that attacks are invisible and unstoppable, the fact that the attacker cannot be detected, and most significantly the problematic implementation of the same origin policy. This subsection proposes several approaches which can be used to address these weaknesses. Some of these defenses "break the standard" and make existing use cases for HbbTV applications (such as tracking cookies) impractical. Other defenses are less disruptive and can be independently deployed by security-minded equipment vendors and even marketed as differentiating features of their TV sets.

7.3.1 Crowdsource detection of RF attacks

Acting alone, an individual television set can do little to detect that its broadcast TV signal is suddenly coming from a malicious source. However, multiple television sets in the same area can aggregate their statuses, making it possible to use this information for detecting radio-based attacks. For example, if the Receive Signal Strength Indication (RSSI) in a certain geographic area has rapidly and suddenly changed, it might mean these TV sets are now receiving a signal from the attacker and not from the original radio tower. The RSSI information can even be used as a form of triangulation, to help pinpoint the exact location of the attacker and aid in his capture. Similarly, if multiple television sets are tuned to the same broadcast frequency, but a certain subset is receiving a different HbbTV application associated with this channel than the other TVs, this can indicate that an attack is in progress. It would be interesting to find a way of achieving this without compromising the privacy of the viewers.

7.3.2 Tighten control over app life cycle

The attacks described here are especially effective since they turn on automatically and without the knowledge of the user, and have no standard way of being disabled. The obvious way of addressing this limitation would be

to guarantee the user's informed consent before active HTML content is rendered by the television. A good analogue to this behavior be found in the WHATWG's recommended implementation of the HTML5 full-screen API [39], which specifies that "User agents should ensure, e.g. by means of an overlay, that the end user is aware something is displayed fullscreen. User agents should provide a means of exiting fullscreen that always works and advertise this to the user." In this spirit, the TV should prompt the user to press the red button **before** rendering any form of HbbTV content for the first time for a given channel, then periodically remind the user that content is running (for example by displaying a brief notification overlay whenever the user switches back to the channel). Users should also have a way of stopping HbbTV rendering for a particular channel.

This countermeasure is perhaps the most intuitive and can be immediatly implemented by individual hardware makers. Sadly, it was shown that users do not react productively to warning messages which interfere with their browsing (or TV watching) [36]. In addition, there are already several established market players who will resist any change to this behaviour, as they already use invisible HbbTV applications for user tracking and analytics.

7.3.3 Prevent broadcast-delivered HTML content from accessing the Internet

It is risky to allow unauthenticated broadcast content to define its own web origin. It seems tempting, then, to create a special restricted origin for broadcasted content, which is distinct from all other Internet domains. Another possible countermeasure is **content signing**. With this proposed defense, all HTML content delivered inside the DVB stream will be accompanied by a signed certificate attesting to its web origin. A malicious adversary cannot sign web pages on behalf of the website under attack, and thus cannot claim these sites as its origin. Unfortunately, even if all broadcast content was properly assigned to a restricted web origin, many attacks would still be possible via "blind" CSRF or PuppetNet attacks [29]. These attacks can cause considerable damage, even if the same-origin principle is upheld, by the sheer virtue of being able to access the Internet using somebody else's computer.

The HbbTV specification conceived the embedding of web content into the DVB data stream as a redundancy method, designed to allow the delivery of interactive content to the 30% of smart TV owners who do not, in fact, plug them into the Internet. This reasoning can be turned into an brutal, but effective, way to secure HbbTV. We recommend to **completely cut off Internet access** to all broadcast-delivered HTML content. Under this model, broadcast-delivered applications will be able to interact only with broadcast-delivered resources, while the only way of getting the television to access the Internet would be through an application delivered in URL form and fetched from the Internet itself. We note that the Google Chrome browser applies a very similar security policy to its browser extensions [17].

7.3.4 Ineffective countermeasures

There are several defensive steps which appear at first to protect against the attack, but whose practical effectiveness is very limited. The first is **content encryption**. Rights-managed DVB content is commonly encrypted, or scrambled, and this encryption appears to be a way of preventing an attack which modifies the television channel. DVB encryption is, however, only applied to individual transport streams such as audio or video. The DVB specification [9] dictates that and not to the program management table (PMT), which points to the HbbTV application, is always sent in the clear. This makes it possible for an adversary to inject a malicious application into any channel, even one with encrypted video and audio.

It will also be inffective to protect against this attack using **Internet proxies**. As suggested by Tews in [16], these "green button" proxies can deliver "sanitised" versions of HbbTV applications to users, after applying modifications which protect the security and privacy of the users. Unfortunately, these proxies are only effective as long as the HbbTV application itself lives on the Internet. Our attack deals with a different form of delivery, where the application resides inside the broadcast television stream.

7.4 Related Work

Works investigating other security issues with Smart TVs were published by Grattafiori and Yavor in [18] and by Lee and Kim in [31]. The first academic work to deal with security weaknesses in HbbTV was published by Tews et al. in [16, 15]. This work focused on potential

privacy leaks resulting from the use of HbbTV. The authors showed how an adversary sniffing encrypted traffic generated by HbbTV on a user's wireless network can infer which program the user is currently watching, even without decrypting the packets. This work also suggests a proxy-based method for blocking autostart applications from running on the television without user permissions.

Another series of works on HbbTV was published by Martin Herfurt [21, 20]. Herfurt surveyed the HTML applications used by German HbbTV providers, discovering that many of them use HbbTV to periodically "phone home" and notify that the user is tuned to the station. Since this was done without the user's consent, these behaviours were considered a breach of German privacy laws. Herfurt additionally suggested a series of attacks which might be possible using HbbTV, including content spoofing, intranet attacks and even bitcoin mining. Finally, Herfurt also implemented a DNS-based privacy protection method called HbbTV Access Limiter (HAL).

Our work significantly contributes to that of of Herfurt and Tews et al. in two aspects. First, our work is the first to present and evaluate a cost-effective method of injecting malicious content into HbbTV systems, by using an RF-based man-in-the-middle attack. Second, our work is the first to call attention to the flawed specification of the same-origin policy for embedded HTML content, and to the devastating cross-domain attacks made possible by this flaw. It is the combination of a feasible attack model and a faulty security model which makes the attacks described in this paper so practical and so dangerous.

The most troubling attacks we discuss result from a flawed implementation of the Same-Origin Policy. As described by Johns et al. in [25], there have been several historical compromises of this policy, starting from 1996 [12], with each compromise resulting in serious consequences for web security. This work can be viewed as a particular instance of this case, made even more powerful due to the broadcast nature of the attack. Our work can also be viewed as a form of **cross-mechanism vulnerability**, in which the combination of perfectly benign broadcast and broadband systems create a system-of-systems with an **emergent property** which allows it to be compromised. Similar properties have previously been demonstrated in voice over IP systems which which combine Internet and PSTN networks [28].

There have been several previous works which exploit a broadcast radio frequency channel to attack a multitude of computers. Notable are the work of Nighswander et al. which attacks GPS software stacks [32], and the work of Checkoway et al. which attacks car computers via the broadcast FM RDS channel [4].

8 Conclusion

We have described a series of novel attacks on Smart TVs – a widely deployed device whose significance in our life is only likely to grow. The key enabling factor of this attack was the fact that the device can render Internet content whose source is outside the Internet. This makes it possible for a physical attacker to cause a large-scale compromise of the Internet. We qualitatively and quantitively demonstrated that the attacks we described can be cost-effectively distributed to many thousands of users, and that they have a large damage potential. The attacks described in this paper are of high significance, not only because of the very large amount of devices which are vulnerable to them, but because they exemplify the complexity of securing systems-of-systems which combine both Internet and non-Internet interfaces. Similar cyber-physical systems will become increasingly more prevalent in the future Internet of Things, making it especially important to analyze the weaknesses in this system, as well as the limitations of its proposed countermeasures.

Acknowledgements: We thank our shepherd Srđan Čapkun, as well as the anonymous reviewers, for their helpful and instructive comments. Erez Waisbard provided valuable information about MPEG internals. This material is based upon work supported by (while author Keromytis was serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Barth. The web origin concept. RFC 6454 (Proposed Standard), December 2011.

[2] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In

Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[3] Armin Büscher and Thorsten Holz. Tracking DDoS attacks: Insights into the business of disrupting the web. In *Proceedings of the 5th USENIX Conference on Large-Scale Exploits and Emergent Threats*, LEET'12, pages 8–8, Berkeley, CA, USA, 2012.

[4] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 6–6, Berkeley, CA, USA, 2011.

[5] Federal Communications Commission. Review of the commission's rules and policies affecting the conversion to digital television. `http://fjallfoss.fcc.gov/edocs_public/attachmatch/FCC-01-24A1.pdf`.

[6] Advanced Television Systems Committee. A/105: ATSC candidate standard – interactive services standard.

[7] BeEF development team. The browser exploitation framework. `http://beefproject.com`.

[8] Avalpa Digital Engineering. Opencaster: the free digital tv software. `http://www.avalpa.com/the-key-values/15-free-software/33-opencaster`.

[9] European Broadcasting Union. Support for use of the DVB scrambling algorithm version 3 within digital broadcasting systems. ETSI TS 100 289 V1.1.1, September 2011.

[10] European Broadcasting Union. Hybrid Broadcast Broadband TV. ETSI TS 102 796 V1.2.1, September 2012.

[11] European Commision. Special eurobarometer 396 – e-communications household survey, 2013. `http://ec.europa.eu/digital-agenda/en/news/special-eurobarometer-396-e-communications-household-survey`.

[12] Edward Felten, Andrew Appel, and David Walker. DNS-based attack on java. `http://sip.cs.princeton.edu/news/dns-spoof.html`.

[13] TV Fool. Online coverage map browser. `https://www.tvfool.com/index.php?option=com_wrapper&Itemid=80`.

[14] Open IPTV Forum. OIPF specification volume 5 – declarative application environment. `http://www.oipf.tv/specifications`.

[15] Marco Ghiglieri, Florian Oswald, and Erik Tews. HbbTV - I Know What You Are Watching. In *13. Deutschen IT-Sicherheitskongresses*. BSI, SecuMedia Verlags-GmbH, May 2013.

[16] Marco Ghiglieri and Erik Tews. A privacy protection system for HbbTV in smart TVs. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE*, January 2014.

[17] Inc. Google. Chrome extensions – content security policy. `http://developer.chrome.com/extensions/contentSecurityPolicy.html`.

[18] Aaron Grattafiori and Josh Yavor. The outer limits: Hacking the samsung smart TV. `https://www.blackhat.com/us-13/briefings.html#Grattafiori`.

[19] Robert "RSnake" Hansen. Stealing mouse clicks for banner fraud, January 2007. `http://ha.ckers.org/blog/20070116/stealing-mouse-clicks-for-banner-fraud/`.

[20] Martin Herfurt. Security Concerns with HbbTV. BerlinSides 0x04 Lightning Talks, May 2013. `http://mherfurt.wordpress.com/2013/06/01/security-concerns-with-hbbtv/`.

[21] Martin Herfurt. Security issues with Hybrid Broadcast Broadband TV. 30'th Chaos Computer Convention, December 2013. `https://events.ccc.de/congress/2013/Fahrplan/events/5398.html`.

[22] Google Inc. Google inc. announces third quarter 2013 result. `http://investor.google.com/pdf/2013Q3_google_earnings_release.pdf`.

[23] International Standards Institute. Information technology – generic coding of moving pictures and associated audio information – part 1: Systems. ISO/IEC 13818-1, May 2013.

[24] International Telecommunication Union. Planning criteria, including protection ratios, for digital terrestrial television services in the VHF/UHF bands. ITU R-REC-BT.1368, February 2014.

[25] Martin Johns, Sebastian Lekies, and Ben Stock. Eradicating DNS rebinding with the extended same-origin policy. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 621–636, Berkeley, CA, USA, 2013.

[26] Martin Johns and Justus Winter. Protecting the intranet against javascript malware and related attacks. In Bernhard Hämmerli and Robin Sommer, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 4579 of *LNCS*, pages 40–59. Springer Berlin Heidelberg, 2007.

[27] Hans-Joachim Kamp. 40 jahre gfu. `http://www.gfu.de/srv/easyedit/_ts_1373472398000/page:home/download/insightstrends/sl_1338454764893/args.link01/de_kamp.pdf`.

[28] A.D. Keromytis. A comprehensive survey of voice over IP security research. *Communications Surveys Tutorials, IEEE*, 14(2):514–537, March 2012.

[29] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 221–234, New York, NY, USA, 2006. ACM.

[30] American Radio Relay League. *2014 ARRL Handbook for Radio Communications*. American Radio Relay League, 91st edition, October 2013.

[31] SeungJin 'Beist' Lee. Hacking, surveilling and deceiving victims on smart TV. `https://www.blackhat.com/us-13/briefings.html#Lee`.

[32] Tyler Nighswander, Brent Ledvina, Jonathan Diamond, Robert Brumley, and David Brumley. Gps software attacks. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 450–461, New York, NY, USA, 2012. ACM.

[33] VideoLAN Organization. VLC media player. `http://www.videolan.org/vlc/index.html`.

[34] Theodore Reed, Joseph Geis, and Sven Dietrich. SkyNET: A 3G-enabled mobile attack drone and stealth botmaster. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 4–4, Berkeley, CA, USA, 2011.

[35] L. Seirup and G. Yetman. U.s. census grids (summary file 3), 2000: Metropolitan statistical areas, 2006.

[36] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *Proceedings of the 18th USENIX Conference on Security*, SEC'09, pages 399–416, Berkeley, CA, USA, 2009.

[37] The Diffusion Group. Connected TVs now present in six of ten US broadband households. `http://tdgresearch.com/connected-tvs-now-present-in-six-of-ten-us-broadband-households/`.

[38] Kurt Thomas, Damon McCoy, Chris Grier, Alek Kolcz, and Vern Paxson. Trafficking fraudulent accounts: The role of the underground market in twitter spam and abuse. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 195–210, Berkeley, CA, USA, 2013.

[39] Anne van Kesteren and Tantek Çelik. Fullscreen API living standard. `http://fullscreen.spec.whatwg.org`.

Security Analysis of a Full-Body Scanner

Keaton Mowery,[*] Eric Wustrow,[†] Tom Wypych,[*] Corey Singleton,[*] Chris Comfort,[*]
Eric Rescorla,[*] Stephen Checkoway,[‡] J. Alex Halderman,[†] Hovav Shacham[*]
[*] *UC San Diego,* [†] *University of Michigan,* [‡] *Johns Hopkins University*

Abstract

Advanced imaging technologies are a new class of people screening systems used at airports and other sensitive environments to detect metallic as well as nonmetallic contraband. We present the first independent security evaluation of such a system, the Rapiscan Secure 1000 full-body scanner, which was widely deployed at airport checkpoints in the U.S. from 2009 until 2013. We find that the system provides weak protection against adaptive adversaries: It is possible to conceal knives, guns, and explosives from detection by exploiting properties of the device's backscatter X-ray technology. We also investigate cyberphysical threats and propose novel attacks that use malicious software and hardware to compromise the the effectiveness, safety, and privacy of the device. Overall, our findings paint a mixed picture of the Secure 1000 that carries lessons for the design, evaluation, and operation of advanced imaging technologies, for the ongoing public debate concerning their use, and for cyberphysical security more broadly.

1 Introduction

In response to evolving terrorist threats, including nonmetallic explosive devices and weapons, the U.S. Transportation Security Administration (TSA) has adopted advanced imaging technology (AIT), also known as whole-body imaging, as the primary passenger screening method at nearly 160 airports nationwide [50]. Introduced in 2009 and gradually deployed at a cost exceeding $1 billion, AIT provides, according to the TSA, "the best opportunity to detect metallic and non-metallic anomalies concealed under clothing without the need to touch the passenger" [48].

AIT plays a critical role in transportation security, and decisions about its use are a matter of public interest. The technology has generated considerable controversy, including claims that the devices are unsafe [40], violate privacy and civil liberties [27, 41], and are ineffective [8, 21]. Furthermore, AIT devices are complex cyberphysical systems — much like cars [23] and implantable medical devices [13] — that raise novel computer security issues. Despite such concerns, neither the manufacturers nor the government agencies that deploy these machines have disclosed sufficient technical details to facilitate rigorous independent evaluation [40], on the grounds that such information could benefit attackers [48]. This lack of transparency has limited the ability of policymakers, experts, and the public to assess contradicting claims.

Figure 1: The Rapiscan Secure 1000 full-body scanner uses backscattered X-rays to construct an image through clothing. Naïvely hidden contraband, such as the handgun tucked into this subject's waistband, is readily visible to the device operator.

To help advance the public debate, we present the first experimental analysis of an AIT conducted independently of the manufacturer and its customers. We obtained a Rapiscan Secure 1000 full-body scanner — one of two AITs widely deployed by the TSA [32] — and performed a detailed security evaluation of its hardware and software. Our analysis provides both retrospective insights into the adequacy of the testing and evaluation procedures that led up to TSA use of the system, and prospective lessons about broader security concerns, including cyberphysical threats, that apply to both current and future AITs.

The Secure 1000 provides a unique opportunity to investigate the security implications of AITs in a manner that allows robust yet responsible public disclosure. Although it was used by the TSA from 2009 until 2013, it has recently been removed from U.S. airports due to changing functional requirements [34]. Moreover, while the Secure 1000 uses backscatter X-ray imaging, current TSA systems are based on a different technology, mil-

limeter waves [11], so many of the attacks we present are not directly applicable to current TSA checkpoints, thus reducing the risk that our technical disclosures will inadvertently facilitate mass terrorism. However, while Secure 1000 units are no longer used in airports, they still are in use at other government facilities, such as courthouses and prisons (see, e.g., [15, 29]). In addition, other backscatter X-ray devices manufactured by American Science and Engineering are currently under consideration for use at airports [34]. To mitigate any residual risk, we have redacted a small number of sensitive details from our attacks in order to avoid providing recipes that would allow an attacker to reliably defeat the screening process without having access to a machine for testing.

In the first part of our study (Section 3), we test the Secure 1000's effectiveness as a physical security system by experimenting with different methods of concealing contraband. While the device performs well against naïve adversaries, fundamental limitations of backscatter imaging allow more clever attackers to defeat it. We show that an adaptive adversary, with the ability to refine his techniques based on experiment, can confidently smuggle contraband past the scanner by carefully arranging it on his body, obscuring it with other materials, or properly shaping it. Using these techniques, we are able to hide firearms, knives, plastic explosive simulants, and detonators in our tests. These attacks are surprisingly robust, and they suggest a failure on the part of the Secure 1000's designers and the TSA to adequately anticipate adaptive attackers. Fortunately, there are simple procedural changes that can reduce (though not eliminate) these threats, such as performing supplemental scans from the sides or additional screening with a magnetometer.

Next, we evaluate the security of the Secure 1000 as a cyberphysical system (Section 4) and experiment with three novel kinds of attacks against AITs that target their effectiveness, safety features, and privacy protections. We demonstrate how malware infecting the operator's console could selectively render contraband invisible upon receiving a "secret knock" from the attacker. We also attempt (with limited success) to use software-based attacks to bypass the scanner's safety interlocks and deliver an elevated radiation dose. Lastly, we show how an external device carried by the attacker with no access to the console can exploit a physical side-channel to capture naked images of the subject being scanned. These attacks are, in general, less practical than the techniques we demonstrate for hiding contraband, and their limitations highlight a series of conservative engineering choices by the system designers that should serve as positive examples for future AITs.

Finally, we attempt to draw broader lessons from these findings (Section 5). Our results suggest that while the Secure 1000 is effective against naïve attackers, it is not able to guarantee either efficacy or privacy when subject to attack by an attacker who is knowledgeable about its inner workings. While some of the detailed issues we describe are specific to the scanner model we tested, the root cause seems to be the failure of the system designers and deployers to think adversarially. This pattern is familiar to security researchers: past studies of voting machines [4], cars [23] and medical devices [13] have all revealed cyberphysical systems that functioned well under normal circumstances but were not secure in the face of attack. Thus, we believe this study reinforces the message that security systems must be subject to adversarial testing before they can be deemed adequate for widespread deployment.

Research safety and ethics. Since the Secure 1000 emits ionizing radiation, it poses a potential danger to the health of scan subjects, researchers, and passers by. Our institutional review board determined that our study did not require IRB approval; however, we worked closely with research affairs and radiation safety staff at the university that hosted our device to minimize any dangers and assure regulatory compliance. To protect passers by, our device was sited in a locked lab, far from the hallway, and facing a thick concrete wall. To protect researchers, we marked a 2 m region around the machine with tape; no one except the scan subject was allowed inside this region while high voltage was applied to the X-ray tube. We obtained a RANDO torso phantom [33], made from a material radiologically equivalent to soft tissue cast over a human skeleton, and used it in place of a human subject for all but the final confirmatory scans. For these final scans we decided, through consultation with our IRB, that only a PI would be used as a scan subject. Experiments involving weapons were conducted with university approval and in coordination with the campus police department and all firearms were unloaded and disabled. We disclosed our security-relevant findings and suggested procedural mitigations to Rapiscan and the Department of Homeland Security ahead of publication.

Online material. Additional resources and the most recent version of this report are available online at `https://radsec.org/`.

2 The Rapiscan Secure 1000

The Secure 1000 was initially developed in the early 1990s by inventor Steven W. Smith [42, 44]. In 1997, Rapiscan Systems acquired the technology [43] and began to produce the Rapiscan Secure 1000. In 2007, the TSA signed a contract with Rapiscan to procure a customized version of the Secure 1000 for deployment in airport passenger screening [47].

We purchased a Rapiscan Secure 1000 from an eBay seller who had acquired it in 2012 at a surplus auction

from a U.S. Government facility located in Europe [17]. The system was in unused condition. It came with operating and maintenance manuals as well as detailed schematics, which were a significant aid to reverse engineering. The system consists of two separate components: the scanner unit, a large enclosure that handles X-ray generation and detection under the control of a special purpose embedded system, and the user console, a freestanding cabinet that contains a PC with a keyboard and screen. The two components are connected by a 12 m cable.

The system we tested is a dual pose model, which means that the subject must turn around in order to be scanned from the front and back in two passes. TSA screening checkpoints used the Secure 1000 single pose model [32], which avoids this inconvenience by scanning from the front and back using a pair of scanner units. Our system was manufactured in about September 2006 and includes EPROM software version 2.1. Documents obtained under the Freedom of Information Act suggest that more recent versions of the hardware and software were used for airport screening [45, 52], and we highlight some of the known differences below. Consequently, we focus our analysis on fundamental weaknesses in the Secure 1000 design that we suspect also affect newer versions. A detailed analysis of TSA models might reveal additional vulnerabilities.

2.1 Backscatter Imaging

X-ray backscatter imaging exploits the unique properties of ionizing radiation to penetrate visual concealment and detect hidden contraband. The physical process which generates backscatter is Compton scattering, in which a photon interacts with a loosely bound or free electron and scatters in an unpredictable direction [7]. Other interactions, such as the photoelectric effect, are possible, and the fraction of photons that interact and which particular effect occurs depends on each photon's energy and the atomic composition of the mass. For a single-element material, the determining factor is its atomic number Z, while a compound material can be modeled by producing an "effective Z," or Z_{eff} [46].

Under constant-spectrum X-ray illumination, the backscattered intensity of a given point is largely determined by the atomic composition of matter at that location, and to a lesser extent its density. Thus, organic materials, like flesh, can be easily differentiated from materials such as steel or aluminum that are made from heavier elements.

The Secure 1000 harnesses these effects for contraband screening by operating as a "reverse camera," as illustrated in Figure 2. X-ray output from a centrally-located tube (operating at 50 kVp and 5 mA) passes through slits in shielding material: a fixed horizontal slit directly in front of a "chopper wheel," a rapidly spinning disk with four radial slits. This results in a narrow, collimated X-ray beam, repeatedly sweeping across the imaging field. During a scan, which takes about 5.7 s, the entire X-ray assembly moves vertically within the cabinet, such that the beam passes over every point of the scene in a series of scan lines.

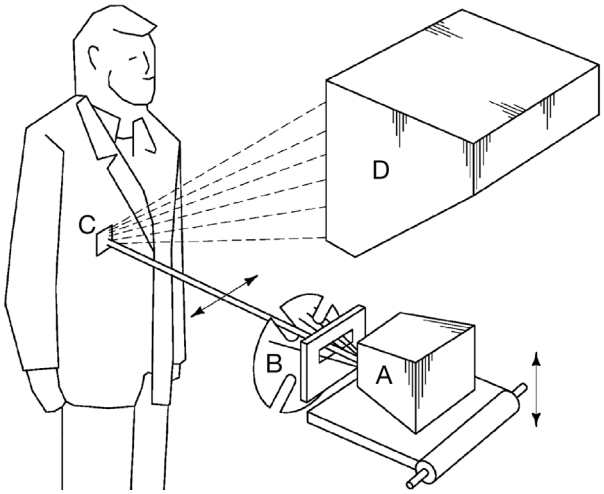


Figure 2: Backscatter Imaging — An X-ray tube (A) mounted on a platform travels vertically within the scanner. The X-rays pass through a spinning disk (B) that shapes them into a horizontally scanning beam. Some photons that strike the target (C) are backscattered toward detectors (D) that measure the reflected energy over time. Adapted from U.S. Patent 8,199,996 [16].

As the beam sweeps across the scene, a set of 8 large X-ray detectors measures the intensity of the backscattered radiation at each point, by means of internal photomultiplier tubes (PMTs). The Secure 1000 combines the output of all 8 detectors, and sends the resulting image signal to the user console, which converts the time-varying signal into a 160×480 pixel monochrome image, with the intensity of each pixel determined by the Z_{eff} value of the surface of the scan subject represented by that pixel location.

2.2 Subsystems

Operator interface. The operator interacts with the Secure 1000 through the user console, a commodity x86 PC housed within a lockable metal cabinet. With our system, the user console is connected to the scanner unit via a serial link and an analog data cable. Documents released by the TSA indicate that airport checkpoint models were configured differently, with an embedded PC inside the scanner unit linked to a remote operator workstation via a dedicated Ethernet network [45, 52].

On our unit, the operator software is an MS-DOS application called SECURE65.EXE that launches automatically when the console boots. (TSA models are apparently Windows-based and use different operator software [45, 47].) This software is written in a BASIC vari-

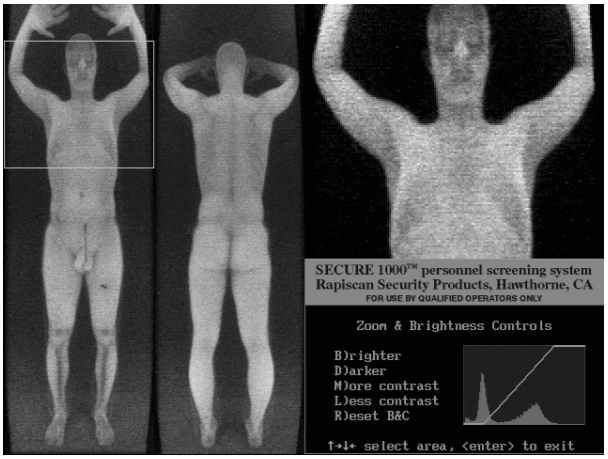


Figure 3: Operator View — The user console displays front and back images and offers basic enhancements and 2 × zoom. It also allows the operator to print images or save them to disk.

ant, and the main user interface is a 640×480 pixel, 4-bit grayscale screen, as shown in Figure 3. The operator invokes a scan by pressing a hand switch. After image acquisition, the operator can inspect the scan by means of a 2× zoom and interactive brightness and contrast controls. The image can also be saved to disk or printed. Further, the software contains several calibration functions that can only be accessed by entering a 4 digit numeric password. The password is hard-coded and is printed in the maintenance manual.

Scanner unit. The scanner unit contains an assortment of electrical and mechanical systems under the control of an embedded computer called the System Control Board (SCB). The SCB houses an Intel N80C196KB12 microcontroller, executing software contained on a 32 KiB socketed ROM. It interacts with the user console PC over a bidirectional RS-232 serial link using simple ASCII commands such as `SU` for "scan up" and `SD` for "scan down." In turn, the SCB uses digital and analog interfaces to direct and monitor other components, including the X-ray tube, PMTs, and chopper wheel. It also implements hardware-based safety interlocks on the production of X-rays, which we discuss further in Section 4.2.

To control vertical movement of the X-ray tube, the scanner unit uses an off-the-shelf reprogrammable servo motor controller, the Parker Gemini GV6. In normal operation, the servo controller allows the SCB to trigger a movement of the X-ray tube, initially to a "home" position and subsequently to scan up and down at predefined rates. There is no command to move the tube to a specific intermediate position.

3 Contraband Detection

As the Secure 1000 is intended to detect prohibited or dangerous items concealed on the body of an attacker, the first and most obvious question to ask is how effectively the Secure 1000 detects contraband.

To make the discussion concrete, we consider the machine as it was typically used by the TSA for airport passenger screening. Under TSA procedures, subjects were imaged from the front and back, but not from the sides. A trained operator inspected the images and, if an anomaly was detected, the passenger was given a manual pat down to determine whether it was a threat [45]. The Secure 1000 was used in place of a walk-through metal detector, rather than both screening methods being employed sequentially [48]. We focus our analysis on threats relevant to an airport security context, such as weapons and explosives, as opposed to other contraband such as illicit drugs or bulk currency.

To replicate a realistic screening environment, we situated our Secure 1000 in an open area, oriented 2.5 m from a concrete wall sufficient to backstop X-ray radiation. This distance accords with the manufacturer's recommendation of at least 2 m of open area "for producing the best possible images" [35]. For typical tests, we arranged the subject at a distance of about 38 cm in front of the scanner using the foot position template provided with the machine.

Naïve adversary. First, we consider the scanner's effectiveness against a naïve adversary, an attacker whose tactics do not change in response to the introduction of the device. Although this is a weak attacker, it seems to correspond to the threat model under which the scanner was first tested by the government, in a 1991 study of a prototype of the Secure 1000 conducted by Sandia National Laboratories [22]. Our results under this threat model generally comport with theirs. Guns, knives, and blocks of explosives naïvely carried on the front or back of the subject's body are visible to the scanner operator.

Three effects contribute to the detectability of contraband. The first is *contrast*: human skin appears white as it backscatters most incident X-ray radiation, while metals, ceramics, and bone absorb X-rays and so appear dark gray or black. The second is *shadows* cast by three-dimensional objects as they block the X-ray beam, which accentuate their edges. The third is *distortion* of the subject's flesh as a result of the weight of the contraband or the mechanics of its attachment. The naïve adversary is unlikely to avoid all three effects by chance.

A successful detection of hidden contraband can be seen in Figure 1. The subject has concealed a .380 ACP pistol within his waistband. The X-ray beam interacts with the gun metal significantly differently than the surrounding flesh, and the sharp contrast in backscatter intensity is immediately noticeable.

Adaptive adversary. Of course, real attackers are not entirely ignorant of the scanner. The TSA announced

(a) Subject with .380 ACP pistol taped above knee.

(b) Subject with .380 ACP pistol sewn to pant leg.

Figure 4: Concealing a Pistol by Positioning — The Secure 1000 cannot distinguish between high Z_{eff} materials, such as a metal handgun, and the absence of a backscatter response. Carefully placed metallic objects can be invisible against the dark background.

that it would be used at screening checkpoints [12, 48], the backscatter imaging mechanism is documented in patents and manufacturer reports [16, 24, 36], images captured with the device have appeared in the media [12, 25], and the physics of backscatter X-rays are well understood [2, 7, 22]. We must assume that attackers have such information and adapt their tactics in response.

To simulate an adaptive adversary, we performed experiments in the style of white-box penetration testing commonly employed in the computer security field. We allowed ourselves complete knowledge of how the scanner operates as well as the ability to perform test scans, observed the resulting images, and used them to adjust our concealment methods.

Such interactive testing is not strictly necessary to develop clever attacks. Indeed, researchers with no access to the Secure 1000 have proposed a number of concealment strategies based only on published information [21], and we experimentally confirm that several of these attacks are viable. However, the ability to perform tests substantially increases the probability that an attack will succeed on the first attempt against a real deployment. A determined adversary might acquire this level of access in several ways: by buying a machine, as we did; by colluding with a dishonest operator; or by probing the security of real installations over time.

In the remainder of this section, we describe experiments with three adaptive concealment techniques and show that they can be used to defeat the Secure 1000. We successfully use them to smuggle firearms, knives, and explosive simulants past the scanner.

3.1 Concealment by Positioning

The first concealment technique makes use of a crucial observation about X-ray physics: backscatter screening machines emitting X-rays in the 50 keV range, such as the Secure 1000, cannot differentiate between the absence of matter and the existence of materials with high Z_{eff} (e.g., iron and lead). That is, when the scanner emits probing X-rays in a direction and receives no backscatter, it can

either be because the beam interacted with nothing, i.e., traveled unimpeded past the screening subject, or because the beam shone directly upon a material which absorbed it entirely and thus did not backscatter. In either case, the resulting pixels will be dark.

These facts lead directly to a straightforward concealment attack for high Z_{eff} contraband: position the object such that it avoids occluding the carrier's body with respect to the X-ray beam. This technique was first suggested on theoretical grounds by Kaufman and Carlson [21]. In limited trials, a TSA critic used it to smuggle small metal objects through airport checkpoints equipped with the Secure 1000 and other AITs [8]. Note that this attack is not enabled by a poor choice of image background color; as discussed above, the scanner cannot differentiate between the metal objects and the absence of material.

To more fully investigate this attack, we obtained a set of weapons: both knives and firearms, ranging from a .380 ACP pistol to an AR-15 semi-automatic rifle. When we scanned the weapons against a dark backdrop, most of the firearms were readily visible due to the presence of nonmetallic parts. After testing a number of firearms, we settled on our .380 ACP pistol as the most suitable candidate for concealment.

We performed several trials to test different placement and attachment strategies. In the end, we achieved excellent results with two approaches: carefully affixing the pistol to the outside of the leg just above the knee using tape, and sewing it inside the pant leg near the same location. Front and back scans for both methods are shown in Figure 4. In each case, the pistol is invisible against the dark background, and the attachment method leaves no other indication of the weapon's presence.

In a similar test, we concealed an 11 cm metal folding knife, in its closed position, along our test subject's side. In this case, too, front and back scans were completely unable to detect the weapon.

Fortunately, simple procedural changes can thwart these attacks. Instead of performing only front and back scans, every subject could also be made to undergo scans from the left and right sides. Under these scans, a high Z_{eff} weapon positioned on the side of the body would be as obvious as the one in Figure 1. Unfortunately, these additional scans would nearly halve the maximum throughput of the checkpoint, as well as double each person's radiation dose. Another possible mitigation would be to screen each subject with a magnetometer, which would unequivocally find metallic contraband but would fail to uncover more exotic weapons, such as ceramic knives [50, 54]. We note that the attacker's gait or appearance might be compromised by the mass and bulk of the firearm or knife, and this might be noticeable to security personnel outside of the backscatter X-ray screening.

3.2 Concealment by Masking

The second object concealment techniques we attempted are similarly based on X-ray physics: the brightness of a material in the image is directly correlated to its backscatter intensity, which in turn is determined by the Z_{eff} and density of the matter in the path of the beam. Therefore, any combination of substances which scatter incoming X-rays at the same approximate intensity as human flesh will be indistinguishable from the rest of the human.

One consequence of this fact is that high-Z_{eff} contraband can be concealed by masking it with an appropriate thickness of low-Z_{eff} material. We experimented with several masking materials to find one with a Z_{eff} value close to that of flesh. We obtained good results with the common plastic PTFE (Teflon), although due to its low density a significant thickness is required to completely mask a metallic object.

To work around this issue, we took advantage of the Secure 1000's ability to see bones close to the skin. Figure 5 demonstrates this approach: an 18 cm knife is affixed to the spine and covered with 1.5 cm of PTFE. As the X-rays penetrate through the material, they backscatter so that the knife outline approximates our subject's spine. While this mask arrangement creates hard edges and shadows which render it noticeable to screening personnel these effects could be reduced by tapering the edges of the mask.

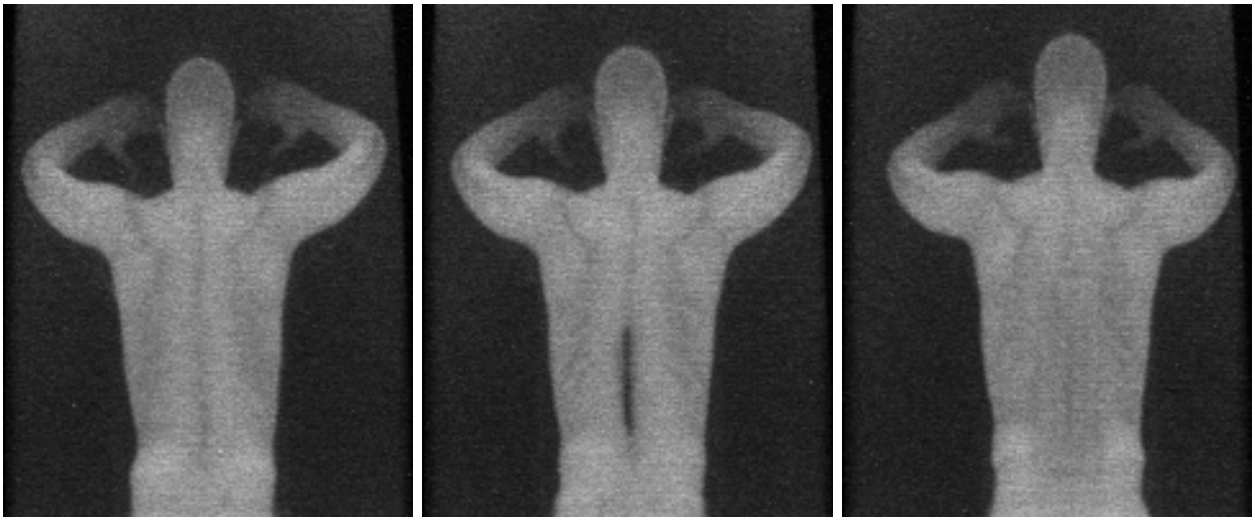
A more difficult challenge for the attacker is taking into account the anatomy of the specific person being imaged. Shallow bones and other dense tissue are visible to the scanner under normal conditions, and a poorly configured mask will stand out against these darker areas of the scan. We conclude that masking can be an effective concealment technique, but achieving high confidence of success would require access to a scanner for testing.

3.3 Concealment by Shaping

Our third and final concealment technique applies a strategy first theorized in [21] to hide malleable, low-Z_{eff} contraband, such as plastic explosives. These materials produce low contrast against human flesh, and, unlike rigid weapons, the attacker can reshape them so that they match the contours of the body.

To experiment with this technique, we acquired radiological simulants for both Composition C-4 [56] and Semtex [57], two common plastic high explosives. These simulants are designed to emulate the plastic explosives with respect to X-ray interactions, and both are composed of moldable putty, similar to the actual explosive materials. We imaged both C-4 and Semtex simulants with the Secure 1000, and found that they appear very similar. We selected the C-4 simulant for subsequent tests.

Our initial plan was to modify the simulants' Z_{eff} to better match that of flesh, by thoroughly mixing in fine metallic powder. To our surprise, however, a thin pancake



(a) No contraband **(b)** 18 cm knife taped to spine **(c)** Knife behind 1.5 cm plastic block

Figure 5: Concealing a Knife by Masking — We find that high-Z_{eff} materials can be hidden by covering them with lower Z_{eff} materials, such as the common plastic PTFE (Teflon). For example, a metal knife is clearly visible when naïvely concealed, but when covered with a thin plastic block it approximates the color of the spine. Tapering the block's edges would reduce the visible outline.

(about 1 cm) of unmodified C-4 simulant almost perfectly approximated the backscatter intensity of our subject's abdomen.

We affixed the pancake with tape (which is invisible to the Secure 1000), and faced two further problems. First, the pancake covered our subject's navel, which is normally clearly visible as a small black area in the scans. Second, by design, plastic explosives are almost completely inert without a matching detonator. These problems neatly solve each other: we attached a detonator, consisting of a small explosive charge in a metal shell, directly over our subject's navel. Since the detonator is coated in metal, it absorbs X-rays quite well and mimics the look of the navel in the final image.

Figure 6 shows a side-by-side comparison of our test subject both carrying no contraband and carrying 200 g of C-4 explosive and attached detonator. To put this amount in perspective, "Shoe Bomber" Richard Reid reportedly carried about 280 g of explosive material [6], and the bomb that destroyed Pan Am Flight 103 is thought to have contained 350 g of Semtex [55].

These scans indicate that plastic explosives can be smuggled through a Secure 1000 screening, since thin pancakes of these materials do not contrast strongly with flesh. While a metal detector would have been sufficient to detect the detonator we used, not all detonators have significant metal components.

In summary, an adaptive adversary can use several attack techniques to carry knives, guns, and plastic explosives past the Secure 1000. However, we also find that multiple iterations of experimentation and adjustment are likely necessary to achieve consistent success. The security of the Secure 1000, then, rests strongly on the adversary's inability to acquire access to the device for testing. However, since we were able to purchase a Secure 1000, it is reasonable to assume that determined attackers and well-financed terrorist groups can do so as well. We emphasize that procedural changes — specifically, performing side scans and supplementing the scanner with a magnetometer — would defeat some, though not all, of the demonstrated attacks.

4 Cyberphysical Attacks

The Secure 1000, like other AITs, is a complex cyberphysical system. It ties together X-ray emitters, detectors, and analog circuitry under the control of embedded computer systems, and feeds the resulting image data to a traditional desktop system in the user console. In this section, we investigate computer security threats against AITs. We demonstrate a series of novel software- and hardware-based attacks that undermine the Secure 1000's efficacy, safety features, and privacy protections.

4.1 User Console Malware

The first threat we consider is malware infecting the user console. On our version of the Secure 1000, the user console is an MS-DOS–based PC attached to the scanner unit via a proprietary cable; TSA models apparently used Windows and a dedicated Ethernet switch [47, 49]. Although neither configuration is connected to an external network, there are several possible infection vectors. If the operators or maintenance personnel are malicious, they could abuse their access in order to manually install malware.

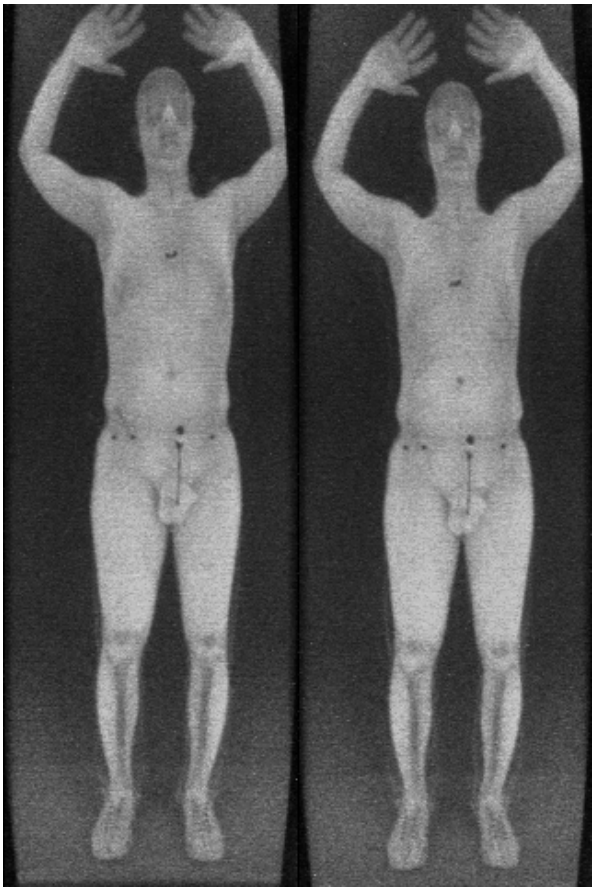


Figure 6: Concealing Explosives by Shaping — *Left:* Subject with no contraband. *Right:* Subject with more than 200 g of C-4 plastic explosive simulant plus detonator, molded to stomach.

The software on our machine lacks any sort of electronic access controls (e.g., passwords) or software verification. While the PC is mounted in a lockable cabinet, we were able to pick the lock in under 10 seconds with a commercially available tool. Therefore, even an outsider with temporary physical access could easily introduce malicious code. TSA systems may be better locked down, but sophisticated adversaries have a track record of infecting even highly secured, airgapped systems [26, 31].

We implemented a form of user console malware by reverse engineering SECURE65.EXE, the front-end software package used by the Secure 1000, and creating a malicious clone. Our version, INSECURE.EXE, is a functional, pixel-accurate reimplementation of the original program and required approximately one man-month to create.

In addition to enabling basic scanning operations, INSECURE.EXE has two malicious features. First, every scan image is saved to a hidden location on disk for later exfiltration. This is a straightforward attack, and it demonstrates one of many ways that software-based privacy protections can be bypassed. Of course, the user could also take a picture of the screen using a camera or smartphone — although operators are forbidden to have such devices in the screening room [39].

Second, INSECURE.EXE selectively subverts the scanner's ability to detect contraband. Before displaying each scan, it applies a pattern recognition algorithm to look for a "secret knock" from the attacker: the concentric squares of a QR code position block. If this pattern occurs, INSECURE.EXE replaces the real scan with a preprogrammed innocuous image. The actual scan, containing the trigger pattern and any other concealed contraband, is entirely hidden.

To trigger this malicious substitution, the subject simply wears the appropriate pattern, made out of any material with a sufficiently different Z_{eff} than human tissue. In our experiments, we arranged lead tape in the target shape, attached to an undershirt, as shown in Figure 7. When worn under other clothing, the target is easily detected by the malware but hidden from visual inspection.

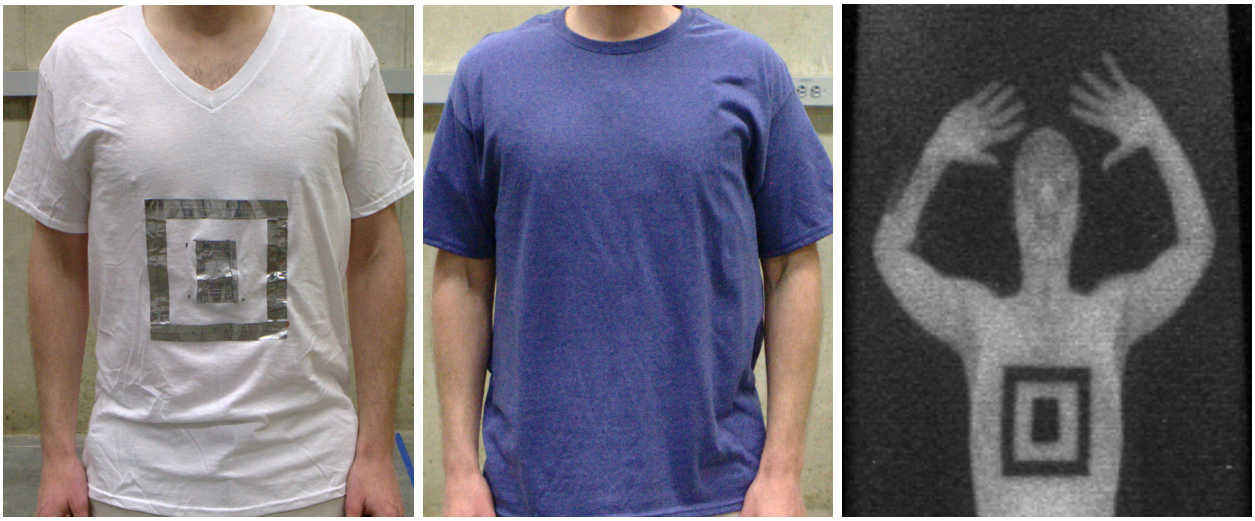
Recently, in response to privacy concerns, the TSA has replaced manual review of images with algorithmic image analysis software known as automated target recognition (ATR) [51]. Instead of displaying an image of the subject, this software displays a stylized figure, with graphical indicators showing any regions which the software considers suspect and needing manual resolution. (Delays in implementing this algorithm led the TSA to remove Secure 1000 machines from airports entirely [1].) If malware can compromise the ATR software or its output path, it can simply suppress these indicators — no image replacement needed.

4.2 Embedded Controller Attacks

The System Control Board (SCB) managing the physical scanner is a second possible point of attack. While the SCB lacks direct control over scan images, it does control the scanner's mechanical systems and X-ray tube. We investigated whether an attacker who subverts the SCB firmware could cause the Secure 1000 to deliver an elevated radiation dose to the scan subject.

This attack is complicated by the fact that the Secure 1000 includes a variety of safety interlocks that prevent operation under unexpected conditions. Circuits sense removal of the front panel, continuous motion of the chopper wheel and the vertical displacement servo, X-ray tube temperature and supply voltage, X-ray production level, key position ("Standby" vs. "On"), and the duration of the scan, among other parameters. If any anomalous state is detected, power to the X-ray tube is immediately disabled, ceasing X-ray emission.

While some of these sensors merely provide inputs to the SCB software, others are tied to hard-wired watchdog circuits that cut off X-ray power without software mediation. However, the firmware can *bypass* these hardware interlocks. At the beginning of each scan, operational



(a) Lead tape pattern **(b)** Pattern concealed **(c)** Scan as captured

Figure 7: A Secret Knock — We demonstrate how malware infecting the Secure 1000 user console could be used to defeat the scanner. The malware is triggered when it detects a specific pattern in a scan, as shown here. It then replaces the real image (c) of the attacker, which might reveal hidden contraband, with an innocuous image stored on disk. Pattern recognition occurs in real time.

characteristics such as tube voltage and servo motion fluctuate outside their nominal ranges. To prevent immediate termination of every scan, SCB software temporarily asserts a bypass signal, which disables the hardware interlocks. This signal feeds a "bypass watchdog" circuit of its own, meant to prevent continual interlock bypass, but the SCB can pet this watchdog by continuously toggling the bypass signal, and cause all hardware interlocks to be ignored. Thus, every safety interlock is either directly under software control or can be bypassed by software.

We developed replacement SCB firmware capable of disabling all of the software and hardware safety interlocks in the Secure 1000. With the interlocks disabled, corrupt firmware can, for instance, move the X-ray tube to a specific height, stop the chopper wheel, and activate X-ray power, causing the machine to deliver the radiation dose from an entire dose to a single point. Only the horizontal displacement of this point is not directly under firmware control — it depends on where the chopper wheel happens to come to rest.

Delivering malicious SCB firmware presents an additional challenge. The firmware is stored on a replaceable socketed EPROM inside the scanner unit, which is secured by an easily picked wafer tumbler lock. Although attackers with physical access could swap out the chip, they could cause greater harm by, say, hiding a bomb inside the scanner. For SCB attacks to pose a realistic safety threat, they would need to be remotely deployable.

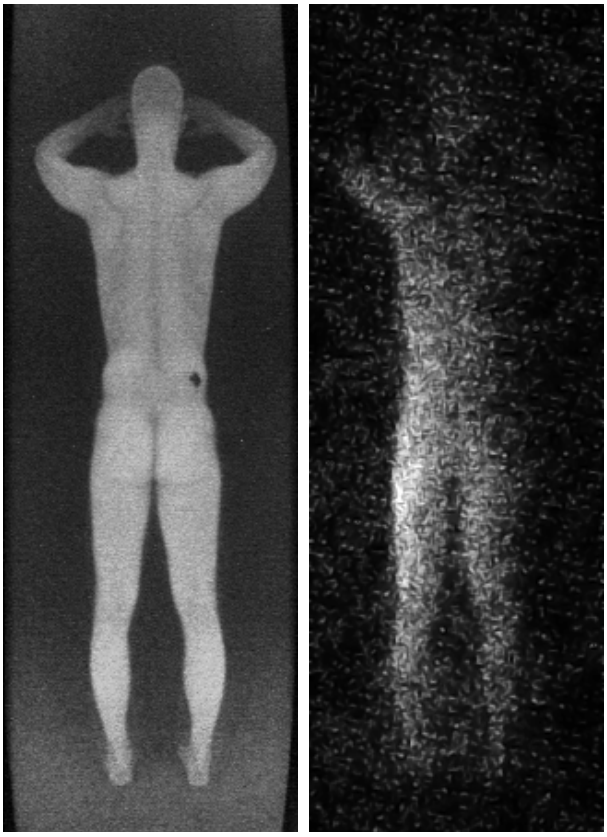
Due to the scanner's modular design, the only feasible vector for remote code execution is the serial link between the user console and the SCB. We reverse engineered the SCB firmware and extensively searched for vulnerabilities. The firmware is simple (<32 KiB) and appears to withstand attacks quite well. Input parsing uses a fixed length buffer, to which bytes are written from only one function. This function implements bounds checking correctly. Data in the buffer is always processed in place, rather than being copied to other locations that might result in memory corruption. We were unable to cause any of this code to malfunction in a vulnerable manner.

While we are unable to remotely exploit the SCB to deliver an elevated radiation dose, the margin of safety by which this attack fails is not reassuring. Hardware interlocks that can be bypassed from software represent a safety mechanism but not a security defense. Ultimately, the Secure 1000 is protected only by its modular, isolated design and by the simplicity of its firmware.

4.3 Privacy Side-Channel Attack

AIT screening raises significant privacy concerns because it creates a naked image of the subject. Scans can reveal sensitive information, including anatomical size and shape of body parts, location and quantity of fat, existence of medical conditions, and presence of medical devices such as ostomy pouches, implants, or prosthetics. As figures throughout the paper show, the resulting images are quite revealing.

Recognizing this issue, the TSA and scanner manufacturers have taken steps to limit access to raw scanned images. Rapiscan and DHS claim that the TSA machines had no capacity to save or store the images [27, 45]. The TSA also stated that the backscatter machines they used had a "privacy algorithm applied to blur the image" [50]. We are unable to verify these claims due to software dif-



(a) From scanner **(b)** From external detector

Figure 8: Attacking Privacy — An attacker could use a detector hidden in a suitcase to capture images of the subject during scanning. As a proof of concept, we used a small external PMT to capture images that are consistent with the scanner's output. A larger detector would produce more detailed images.

ferences between our machine and TSA models. Our Secure 1000 has documented save, recall (view saved images), and print features and does not appear to have a mechanism to disable them. In fact, using forensic analysis software on the user console's drive, we were able to recover a number of stored images from test scans that were incompletely deleted during manufacturing.

These software-based defenses aim to safeguard privacy in images that are constructed by the machine, but they do not address a second class of privacy attacks against AITs: an outsider observer could try to reconstruct scanned images by using their own external detector hardware. The most mechanically complex, dangerous, and energy intensive aspects of backscatter imaging are related to X-ray illumination; sensing the backscattered radiation is comparatively simple. Since X-rays scatter off the subject in a broad arc, they create a kind of physical side channel that potentially leaks a naked image of the subject to any nearby attacker. To the best of our knowledge, we are the first to propose such an attack; the privacy threat model for AITs appears to have been focused almost entirely on concerns about the behavior of screening personnel, rather than the general public.

In the scenario we envision, an attacker follows a target subject (for instance, a celebrity or politician) to a screening checkpoint while carrying an X-ray detector hidden in a suitcase. As the victim is scanned, the hardware records the backscattered X-rays for later reconstruction.

We experimented with the Secure 1000 to develop a proof-of-concept of such an attack. The major technical challenge is gathering enough radiation to have an acceptable signal/noise ratio. The Secure 1000 uses eight large photomultiplier tubes (PMTs) — four on either side of the X-ray generator — in order to capture as much signal as possible. For best results, an attacker should likewise maximize observing PMT surface area, and minimize distance from the subject, as radiation intensity falls off quadratically with distance. To avoid arousing suspicion, an attacker may be limited to only one PMT, and may also be restricted in placement.

To determine whether external image reconstruction is feasible, we used a small PMT, a 75 mm Canberra model BIF2996-2 operated at 900 V, with a 10 cm $\times$ 10 cm NaI crystal scintillator. We placed this detector adjacent to the scanner and fed the signal to a Canberra Model 1510 amplifier connected to a Tektronix DPO 3014 oscilloscope. After capturing the resulting signal, we converted the time varying intensity to an image and applied manual enhancements to adjust levels and remove noise.

Figure 8 shows the results from the scanner and from our corresponding reconstruction. While our proof-of-concept results are significantly less detailed than the scanner's output, they suggest that a determined attacker, equipped with a suitcase-sized PMT, might achieve satisfactory quality. A further concern is that changes in future backscatter imaging devices might make this attack even more practical. Since the PMTs in the Secure 1000 are close to the maximum size that can fit in the available space, further improvements to the scanner's performance — i.e., better resolution or reduced time per scan — would likely require increased X-ray output. This would also increase the amount of information leaked to an external detector.

5 Discussion and Lessons

The Secure 1000 appears to perform largely as advertised in the non-adversarial setting. It readily detected a variety of naïvely concealed contraband materials. Our preliminary measurements of the radiation exposure delivered during normal scanning (Appendix A) seem consistent with public statements by the manufacturer, TSA, and the FDA [5, 18, 38, 54]. Moreover, it seems clear that the manufacturer took significant care to ensure that predictable equipment malfunctions would not result in un-

safe radiation doses; in order for this to happen a number of independent failures would be required, including failures of safety interlocks specifically designed to prevent unsafe conditions.

However, the Secure 1000 performs less well against clever and adaptive adversaries, who can use a number of techniques to bypass its detection capabilities and to attempt to subvert it by cyberphysical means. In this section, we use the device's strengths and weaknesses to draw lessons that may help improve the security of other AITs and cyberphysical security systems more generally.

The effectiveness of the device is constrained by facts of X-ray physics . . . As discussed in Section 2.1, Compton scattering is the physical phenomenon which enables backscatter imaging. As the tight beam of X-rays shines upon the scene, it interacts with the scene material. The intensity and energy spectrum of the backscattered radiation is a function of both the X-ray spectrum emitted by the imaging device and the atomic composition of the material in the scene.

The Secure 1000 emits a single constant X-ray spectrum, with a maximum energy of 50 keV, and detects the intensity of backscatter to produce its image. Any two materials, no matter their actual atomic composition, that backscatter the same approximate intensity of X-rays will appear the same under this technology. This physical process enables our results in Section 3.3. This issue extends beyond the Secure 1000: any backscatter imaging device based upon single-spectrum X-ray emission and detection will be vulnerable to such attacks.

By contrast, baggage screening devices (such as the recently studied Rapiscan 522B; see [37]) usually use transmissive, rather than backscatter, X-ray imaging. These devices also often apply dual-energy X-ray techniques that combine information from low-energy and high-energy scans into a single image. To avoid detection by such systems, contraband will need to resemble benign material under two spectra, a much harder proposition.

. . . but physics is irrelevant in the presence of software compromise. In the Secure 1000, as in other cyberphysical screening systems, the image of the object scanned is processed by software. If that software has been tampered with, it can modify the actual scan in arbitrary ways, faking or concealing threats. Indeed, the ability of device software to detect threats and bring them to the attention of the operator is presumed in the "Automated Target Recognition" software used in current TSA millimeter-wave scanners [51]. Automatic suppression of threats by malicious software is simply the (easier to implement) dual of automatic threat detection. As we show in Section 4.1, malware can be stealthy, activating only when it observes a "secret knock."

Software security, including firmware updates, networked access, and chain-of-custody for any physical media, must be considered in any cyberphysical scanning system. Even so, no publicly known study commissioned by TSA considers software security.

Procedures are critical, but procedural best practices are more easily lost than those embedded in software. As early as 1991, Sandia National Labs recommended the use of side scans to find some contraband:

> A metallic object on the side of a person would blend in with the background and be unobserved. However, a side scan would provide an image of the object. There are other means of addressing this which IRT is considering presently [22, page 14].

Yet TSA procedures appear to call for only front and back scans, and the device manual characterizes side scans as an unusual practice:

> The Secure 1000 can conduct scans in four subject positions, front, rear, left side and right side. Most users only conduct front and rear scans in routine operations and reserve the side scans for special circumstances [35, page 3-7].

Omitting side scans makes it possible to conceal firearms, as we discuss in Section 3.1.

Since side scans are necessary for good security, the device's design should encourage their use by default. Yet, if anything, the scanner user interface nudges operators away from performing side scans. It allows the display of only two images at a time, making it poorly suited to taking four scans of a subject. A better design would either scan from all sides automatically (the Secure 1000 is already sold in a configuration that scans from two sides without the subject's turning around) or encourage/require a four-angle scan.

Adversarial thinking, as usual, is crucial for security. The Sandia report concludes that both C-4 and Detasheet plastic explosives are detected by the Secure 1000. Attached to their report is an image from one C-4 test (Figure 9), wherein a 0.95 cm thick C-4 block is noticeable only by edge effects — it is outlined by its own shadow, while the intensity within the block almost exactly matches the surrounding flesh. This suggests a failure to think adversarially: since plastic explosives are, by design, moldable putty, the attacker can simply gradually thin and taper the edges of the mass, drastically reducing edge effects and rendering it much less noticeable under X-ray backscatter imaging. We describe precisely such an attack in Section 3.3.

The basic problem appears to be that the system, while well engineered, appears not to have been designed, documented, or deployed with adaptive attack in mind. For

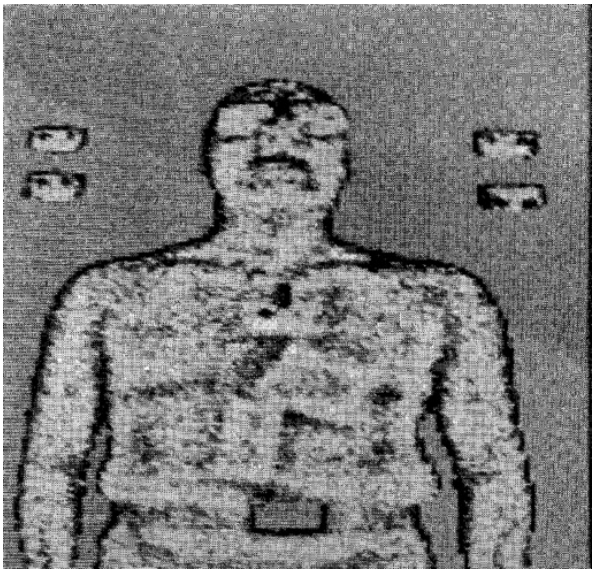


Figure 9: Naïve Evaluation — In an evaluation by Sandia National Labs, a Secure 1000 prototype successfully detects blocks of C-4 plastic explosive and Lucite attached to the subject's chest. Observe that the detection is based almost entirely on the X-ray shadow surrounding each rectangular block, which can be reduced or eliminated by an adaptive adversary through clever shaping and positioning of contraband. Reproduced from [22].

instance, attaching contraband to the side of the body as described in Section 3.1 is a straightforward attack that is enabled by scanning only straight-on rather than from all angles. However, the operator's manual shows only example images where the contraband is clearly at the front or the back.

The other attacks we describe in Sections 3 and 4, which allow us to circumvent or weaken the advertised efficacy, privacy, and security claims, again show that the system's designers failed to think adversarially.

Simplicity and modular design are also crucial for security. The system control board implements simple, well-defined functionality and communicates with the operator console by means of a simple protocol. We were unable to compromise the control board by abusing the communication protocol. This is in contrast to the scanner console, whose software runs on a general-purpose COTS operating system.

Simplicity and modular design prevented worse attacks, but do other AITs reflect these design principles? Modern embedded systems tend towards greater integration, increased software control, and remote network capabilities, which are anathema to security.

Components should be designed with separation of concerns in mind: each component should be responsible for controlling one aspect of the machine's operation. Communication between components should be constrained to narrow data interfaces. The Secure 1000 gets these principles right in many respects. For example, the PC software does not have the ability to command the X-ray tube to a particular height. Instead, it can only command the tube to return to its start position or to take a scan.

Our main suggestion for improving the Secure 1000's cyberphysical security is to remove the ability for the control board firmware to override the safety interlocks (something currently needed only briefly, at scan initialization). As long as this bypass functionality is in place, the interlocks can serve as safety mechanisms but not as a defense against software- or firmware-based attacks.

Keeping details of the machine's behavior secret didn't help . . . Published reports about the Secure 1000 have been heavily redacted, omitting even basic details about the machine's operation. This did not stop members of the public from speculating about ways to circumvent the machine, using only open-source information. In an incident widely reported in the press, Jonathan Corbett suggested that firearms hanging off the body might be invisible against the dark background [8], an attack we confirm and refine in Section 3.1. Two physicists, Leon Kaufman and Joseph Carlson, reverse engineered the Secure 1000's characteristics from published scans and concluded that "[i]t is very likely that a large (15–20 cm in diameter), irregularly-shaped, [one] cm-thick pancake [of plastic explosive] with beveled edges, taped to the abdomen, would be invisible to this technology" [21], an attack we confirm and refine in Section 3.3. Keeping basic information about the device secret made an informed public debate about its use at airports more difficult, but did not prevent dangerous attacks from being devised.

. . . but keeping attackers from testing attacks on the machine might. To a degree that surprised us, our attacks benefited from testing on the device itself. Our first attempts at implementing a new attack strategy were often visible to the scanner, and reliable concealment was made possible only by iteration and refinement. It goes without saying that software-replacement attacks on the console are practical only if one has a machine to reverse engineer. As a result, we conclude that, in the case of the Secure 1000, keeping the machine out of the hands of would-be attackers may well be an effective strategy for preventing reliable exploitation, even if the details of the machine's operation were disclosed.

The effectiveness of such a strategy depends critically on the difficulty of obtaining access to the machine. In addition to the device we purchased, at least one other Secure 1000 was available for sale on eBay for months after we obtained ours. We do not know whether it sold, or to whom. Also, front-line security personnel will always have some level of access to the device at each deployment

installation (including at non-TSA facilities) as they are responsible for its continued operation. Given these facts, imposing stricter purchase controls on backscatter X-ray machines than those currently enacted may not be enough to keep determined adversaries from accessing, studying, and experimenting with them.

6 Related work

Cyberphysical devices must be evaluated not only for their safety but also for their security in the presence of an adversary [19]. This consideration is especially important for AITs, which are deployed to security checkpoints. Unfortunately, AIT manufacturers and TSA have not, to date, allowed an unfettered independent assessment of AITs. Security evaluators retained by a manufacturer or its customers may not have an incentive to find problems [30]. In the case of a backscatter X-ray AIT specifically, an evaluation team may be skilled in physics but lack the expertise to identify software vulnerabilities, or vice versa.

Ours is the first study to consider computer security aspects of an AIT's design and operation, and the first truly independent assessment of an AIT's security, privacy, and efficacy implications informed by experimentation with an AIT device.

Efficacy and procedures. In 1991, soon after its initial development, the Secure 1000 was evaluated by Sandia National Laboratories on behalf of IRT Corp., the company then working to commercialize the device. The Sandia report [22] assessed the device's effectiveness in screening for firearms, explosives, nuclear materials, and drugs. The Sandia evaluators do not appear to have considered adaptive strategies for positioning and shaping contraband, nor did they consider attacks on the device's software. Nevertheless, they observed that side scans were sometimes necessary to detect firearms.

More recently, the Department of Homeland Security's Office of Inspector General released a report reviewing TSA's use of the Secure 1000 [10]. This report proposed improvements in TSA procedures surrounding the machines but again did not consider adversarial conditions or software vulnerabilities.

Working only from published descriptions of the device, researchers have hypothesized that firearms can be concealed hanging off the body [8] and that plastic explosives can be caked on the body [21]. We confirm these attacks are possible in Section 3 and refine them through access to the device for testing.

Health concerns. The ionizing radiation used by the Secure 1000 poses at least potential health risks. Studies performed on behalf of TSA by the Food and Drug Administration's Center for Devices and Radiological Health [5] and by the Johns Hopkins University Applied Physics Laboratory [18] attempted to quantify the overall radiation dose delivered by the device. Both studies saw public release only in heavily redacted form, going so far as to redact even the effective current of the X-ray tube.

In 2010, Professors at the University of California, San Francisco wrote an open letter to John P. Holdren, the Assistant to the President for Science and Technology, expressing their concern about potential health effects from the use of backscatter X-ray scanners at airports [40]. The letter writers drew on their radiological expertise, but did not have access to a Secure 1000 to study. The FDA published a response disputing the technical claims in the UCSF letter [28], as did the inventor of the Secure 1000, Steven W. Smith [43]. Under dispute was not just the total radiation dose but its distribution through the skin and body. In independent work concurrent with ours, a task group of the American Association of Physicists in Medicine [2] explicitly considered skin dose. The task group's measurements are within an order of magnitude of our own, presented in Appendix A.

7 Conclusion

We obtained a Rapiscan Secure 1000 and evaluated its effectiveness for people screening. Ours was the first analysis of an AIT that is independent of the device's manufacturer and its customers; the first to assume an adaptive adversary; and the first to consider software as well as hardware. By exploiting properties of the Secure 1000's backscatter X-ray technology, we were able to conceal knives, firearms, plastic explosive simulants, and detonators. We further demonstrated that malicious software running on the scanner console can manipulate rendered images to conceal contraband.

Our findings suggest that the Secure 1000 is ineffective as a contraband screening solution against an adaptive adversary who has access to a device to study and to use for testing and refining attacks. The flaws we identified could be partly remediated through changes to procedures: performing side scans in addition to front and back scans, and screening subjects with magnetometers as well as backscatter scanners; but these procedural changes will lengthen screening times.

Our findings concerning the Secure 1000 considered as a cyberphysical device are more mixed. Given physical access, we were able to replace the software running on the scanner console, again allowing attackers to smuggle contraband past the device. On the other hand, we were unable to compromise the firmware on the system control board, a fact we attribute to the separation of concerns embodied in, and to the simplicity of, the scanner design.

The root cause of many of the issues we describe seems to be failure of the system designers to think adversarially. That failure extends also to publicly available evaluations of the Secure 1000's effectiveness. Additionally, the secrecy surrounding AITs has sharply lim-

ited the ability of policymakers, experts, and the general public to assess the government's safety and security claims.

Despite the flaws we identified, we are not able to categorically reject TSA's claim that AITs represent the best available tradeoff for airport passenger screening. Hardened cockpit doors may mitigate the hijacking threat from firearms and knives; what is clearly needed, with or without AITs, is a robust means for detecting explosives. The millimeter-wave scanners currently deployed to airports will likely behave differently from the backscatter scanner we studied. We recommend that those scanners, as well as any future AITs — whether of the millimeter-wave or backscatter [34] variety — be subjected to independent, adversarial testing, and that this testing specifically consider software security.

Acknowledgments

We are grateful to the many people who contributed to the preparation of this report, including Michael Bailey, Nate Cardozo, Cindy Cohn, Prabal Dutta, Jennifer Granick, Nadia Heninger, Daniel Johnson, Daniel Kane, Brian Kantor, Falko Kuester, Sarah Meiklejohn, Kurt Opsahl, Stefan Savage, Daniel Scanderberg, Alex Snoeren, Geoff Voelker, Kai Wang, and the anonymous reviewers. We also thank the outstanding staff members at UCSD and the University of Michigan who helped us navigate the legal and regulatory landscape and made studying the device possible. This material is based in part upon work supported by the National Science Foundation Graduate Research Fellowship Program.

References

[1] M. M. Ahlers. TSA removing "virtual strip search" body scanners. CNN, Jan. 2013. http://www.cnn.com/2013/01/18/travel/tsa-body-scanners.

[2] American Association of Physicists in Medicine. Radiation dose from airport scanners. Technical Report 217, June 2013. http://www.aapm.org/pubs/reports/RPT_217.pdf.

[3] American National Standards Institute. Radiation safety for personnel security screening systems using X-ray or gamma radiation. ANSI/HPS N43.17-2009, Aug. 2009.

[4] D. Bowen et al. "Top-to-Bottom" Review of voting machines certified for use in California. Technical report, California Secretary of State, 2007. http://sos.ca.gov/elections/elections.vsr.htm.

[5] F. Cerra. Assessment of the Rapiscan Secure 1000 body scanner for conformance with radiological safety standards, July 2006. http://www.tsa.gov/sites/default/files/assets/pdf/research/rapiscan_secure_1000.pdf.

[6] CNN. Shoe bomb suspect to remain in custody. CNN, Dec. 2001. http://edition.cnn.com/2001/US/12/24/investigation.plane/.

[7] A. H. Compton. A quantum theory of the scattering of X-rays by light elements. *Physical Review*, 21(5):483, 1923.

[8] J. Corbett. $1B of TSA nude body scanners made worthless by blog: How anyone can get anything past the scanners, Mar. 2012. http://tsaoutofourpants.wordpress.com/2012/03/06/1b-of-nude-body-scanners-made-worthless-by-blog-how-anyone-can-get-anything-past-the-tsas-nude-body-scanners.

[9] J. Danzer, C. Dudney, R. Seibert, B. Robison, C. Harris, and C. Ramsey. Optically stimulated luminescence of aluminum oxide detectors for radiation therapy quality assurance. *Medical Physics*, 34(6):2628, July 2007.

[10] Department of Homeland Security, Office of Inspector General. Transportation Security Administration's use of backscatter units. Technical Report OIG-12-38, Feb. 2012. http://www.oig.dhs.gov/assets/Mgmt/2012/OIG_12-38_Feb12.pdf.

[11] Department of Homeland Security, Science and Technology Directorate. Compilation of emission safety reports on the L3 Communications, Inc. ProVision 100 active millimeter wave advanced imaging technology (AIT) system. Technical Report DHS/ST/TSL-12/118, Sept. 2012. http://epic.org/foia/dhs/bodyscanner/appeal/Emission-Safety-Reports.pdf.

[12] EPIC. Transportation agency's plan to x-ray travelers should be stripped of funding, June 2005. http://epic.org/privacy/surveillance/spotlight/0605/.

[13] K. Fu. Trustworthy medical device software. In *Public Health Effectiveness of the FDA 510(k) Clearance Process: Measuring Postmarket Performance and Other Select Topics: Workshop Report*, July 2011.

[14] M. E. Hoppe and T. G. Schmidt. Estimation of organ and effective dose due to Compton backscatter security scans. *Medical Physics*, 39(6):3396–3403, 2012.

[15] J. Hubbard. New jail to have x-ray scanner used for security at airport. Wilkes Journal-Patriot, Oct. 2013. http://www.journalpatriot.com/news/article_95a398bc-368d-11e3-99ec-0019bb30f31a.html.

[16] R. Hughes. Systems and methods for improving directed people screening, June 12 2012. US Patent 8,199,996.

[17] ivw-agne. Rapiscan Secure 1000 DP (Dual Pose) backscatter body scanner / nacktscanner. eBay listing, 2012. http://www.ebay.com/itm/Rapiscan-Secure-1000-DP-Dual-Pose-Backscatter-Body-Scanner-Nacktscanner-/110999548627.

[18] Johns Hopkins University Applied Physics Laboratory. Radiation safety engineering assessment report for the Rapiscan Secure 1000 in single pose configuration. Technical Report NSTD-09-1085, version 2, Aug. 2010. http://www.tsa.gov/sites/default/files/assets/pdf/research/jh_apl_v2.pdf.

[19] R. G. Johnston. Adversarial safety analysis: Borrowing the methods of security vulnerability assessments. *J. Safety Research*, 35(3):245–48, 2004.

[20] P. A. Jursinic and C. J. Yahnke. In vivo dosimetry with optically stimulated luminescent dosimeters, OSLDs, compared to diodes; the effects of buildup cap thickness and fabrication material. *Medical Physics*, 38(10):5432, 2011.

[21] L. Kaufman and J. W. Carlson. An evaluation of airport X-ray backscatter units based on image characteristics. *Journal of Transportation Security*, 4(1):73–94, 2011.

[22] B. Kenna and D. Murray. Evaluation tests of the SECURE 1000 scanning system. Technical Report SAND 91-2488, UC-830, Sandia National Laboratories, Apr. 1992.

[23] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proc. 31st IEEE Symposium on Security and Privacy*, pages 447–62, May 2010.

[24] A. Kotowski and S. Smith. X-ray imaging system with active detector, Dec. 16 2003. US Patent 6,665,373.

[25] D. Kravets. Court oks airport body scanners, rejects constitutional challenge, July 2011. http://www.wired.com/threatlevel/2011/07/court-approves-body-scanners/.

[26] R. Langner. To kill a centrifuge: A technical analysis of what stuxnet's creators tried to achieve. Online: http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf, Nov. 2013.

[27] A. Lowrey. My visit to the offices of Rapiscan, which makes airport scanners. Slate, Nov. 2010. http://www.slate.com/articles/business/moneybox/2010/11/corporate_junket.html.

[28] J. L. McCrohan and K. R. Shelton Waters. Response to UCSF regarding their letter of concern [40], Oct. 2010. http://www.fda.gov/Radiation-EmittingProducts/RadiationEmittingProductsandProcedures/SecuritySystems/ucm231857.htm.

[29] A. Medici. Guess where TSA's invasive scanners are now? Federal Times, May 2014. http://www.federaltimes.com/article/20140516/DHS/305160012/Guess-where-TSA-s-invasive-scanners-now-.

[30] S. J. Murdoch, M. Bond, and R. Anderson. How certification systems fail: Lessons from the Ware report. *IEEE Security & Privacy*, 10(6):40–44, Nov–Dec 2012.

[31] National Security Agency. Cottonmouth-iii. *Der Spiegel*, 2013. http://www.spiegel.de/international/world/a-941262.html, fetched 2014-02-27.

[32] OSI Systems. OSI Systems receives $25m order from U.S. Transportation Security Administration for advanced imaging technology. Press release, Oct. 2009. http://investors.osi-systems.com/releasedetail.cfm?ReleaseID=413032.

[33] Phantom Laboratory. The rando phantom, ran100 and ran110, 2006. http://www.phantomlab.com/library/pdf/rando_datasheet.pdf.

[34] J. Plungis. Naked-image scanners to be removed from U.S. airports. Bloomberg News, Jan. 2013. http://www.bloomberg.com/news/2013-01-18/naked-image-scanners-to-be-removed-from-u-s-airports.html.

[35] *Secure 1000 Personnel Scanner Operator's Manual*. Rapiscan Systems, Aug. 2005.

[36] Rapiscan Systems. Rapiscan secure 1000, 2005. http://epic.org/privacy/surveillance/spotlight/0605/rapiscan.pdf.

[37] Rapiscan Systems. Rapiscan 522b, 2006. http://www.wired.com/images_blogs/threatlevel/2014/02/RapiScan-522B.pdf, fetched 2014-02-27.

[38] Rapiscan Systems. Rapiscan Secure 1000 health and safety fact sheet, 2012. http://www.rapiscansystems.com/extranet/downloadFile/24_Rapiscan%20Secure%201000-Health%20and%20Safety-Fact%20Sheet.pdf.

[39] G. D. Rossides. TSA Reply to Rep. Bennie G. Thompson, Feb. 2010. http://epic.org/privacy/airtravel/backscatter/TSA_Reply_House.pdf.

[40] J. Sedat, D. Agard, M. Shuman, and R. Stroud. UCSF letter of concern, Apr. 2010. http://www.npr.org/assets/news/2010/05/17/concern.pdf.

[41] A. Shahid. Feds admit they stored body scanner images, despite TSA claim the images cannot be saved, Aug. 2010. http://www.nydailynews.com/news/national/feds-admit-stored-body-scanner-images-tsa-claim-images-saved-article-1.200279.

[42] S. W. Smith. Secure 1000, concealed weapon detection system, 1998. http://www.dspguide.com/secure.htm, fetched 2014-02-27.

[43] S. W. Smith. Re: Misinformation on airport body scanner radiation safety, Dec. 2010. http://tek84.com/downloads/radiation-bodyscanner.pdf.

[44] S. W. Smith. Resume, 2014. http://www.dspguide.com/resume.htm, fetched 2014-02-27.

[45] C. Tate. Privacy impact assessment for the Secret Service use of advanced imaging technology, Dec. 2011. http://epic.org/foia/dhs/usss/Secret-Service-Docs-1.pdf.

[46] M. Taylor, R. Smith, F. Dossing, and R. Franich. Robust calculation of effective atomic numbers: The Auto-Zeff software. *Medical Physics*, 39(4):1769, 2012.

[47] Transportation Security Administration. Contract with rapiscan security products, June 2007. IDV ID: HSTS04-07-D-DEP344, http://epic.org/open_gov/foia/TSA_Rapiscan_Contract.pdf.

[48] Transportation Security Administration. Passenger screening using advanced imaging technology: Notice of proposed rulemaking. *Federal Register*, 78(58):18287–302, Mar. 2013.

[49] Transportation Security Administration Office of Security Technology. Procurement specification for whole body imager devices for checkpoint operations. TSA, Sept. 2008. http://epic.org/open_gov/foia/TSA_Procurement_Specs.pdf.

[50] TSA. AIT: Frequently Asked Questions, May 2013. http://www.tsa.gov/ait-frequently-asked-questions. Fetched May 17, 2013: https://web.archive.org/web/20130517152631/http://www.tsa.gov/ait-frequently-asked-questions.

[51] TSA Press Office. TSA takes next steps to further enhance passenger privacy, July 2011. http://www.tsa.gov/press/releases/2011/07/20/tsa-takes-next-steps-further-enhance-passenger-privacy.

[52] TÜV SÜD America. EMC test report: Secure 1000 WBI, Feb. 2009. http://epic.org/privacy/body_scanners/EPIC_TSA_FOIA_Docs_09_09_11.pdf, pages 162–323.

[53] U.S. Army Institute of Public Health. Rapiscan Secure 1000 Single Pose dosimetry study. Technical report, Jan. 2012. http://www.tsa.gov/sites/default/files/assets/pdf/foia/final_ait_dosimetry_study_report_opa.pdf.

[54] U.S. Department of Homeland Security Office of Health Affairs. Fact sheet: Advanced imaging technology (ait) health & safety, 2010. http://www.oregon.gov/OBMI/docs/TSA-AIT_ScannerFactSheet.pdf.

[55] C. Wain. Lessons from lockerbie. BBC, Dec. 1998. http://news.bbc.co.uk/2/hi/special_report/1998/12/98/lockerbie/235632.stm.

[56] XM Materials. Material safety data sheet XM-03-X (Comp C-4 explosive simulant), Oct. 1999. http://www.xm-materials.com/MSDS/xray_msds/msdsxm_03_x.pdf.

[57] XM Materials. Material safety data sheet XM-04-X (Semtex explosive simulant), Oct. 1999. http://www.xm-materials.com/MSDS/xray_msds/msdsxm_04_x.pdf.

A Radiation Dose Assessment

The Secure 1000 generates low-energy X-rays (50 kVp at 5 mA tube accelerating potential) to construct its images. Although this output is low, the machine still produces ionizing radiation, and careful assessment is necessary to ensure public safety.

The imparted dose has been scrutinized recently by various agencies applying a number of experimental designs [2, 14, 53]. These findings have been consistent with manufacturer claims [38] that per-scan radiation exposure to subjects is nonzero, but is near natural background levels. Additionally, there have been claims and counterclaims surrounding the distribution of dose within the body, with some groups raising concerns that the scanner might impart a minimal deep dose but an overly large skin dose to the subject [5, 40, 43].

To shed light on this question, we executed a brief assessment of the radiological output of the scanner using Landauer Inc.'s InLight whole body dosimeters. These dosimeters give a shallow dose equivalent (SDE), a deep dose equivalent (DDE), and an eye lens dose equivalent. They are analyzed using optically stimulated luminescence (OSL), an established dosimeter technology [9, 20]. We read the results using Landauer's proprietary MicroStar dosimeter reader.

We used a simple experimental design to quantify the dose output: we arranged 21 dosimeters on a RANDO chest phantom positioned upright on a wooden table with a neck-to-floor distance of 144 cm and a source-to-detector distance of 66 cm, approximating the conditions of a normal scan. The dosimeters give a more accurate dose representation if the incident beam is perpendicular to the detector material. In this case, the dosimeters were attached to the chest phantom without regard for beam angle, and so no correction factors were implemented; geometry issues were expected in the results.

The InLight dosimeters require a total dose of at least 50 µSv to be accurate. To irradiate them sufficiently, we performed 4033 consecutive single scans in the machine's normal operating mode. (Each screening consists of at least two such scans: one front and one rear.) A scan was automatically triggered every 12 s and lasted 5.7 s, for a total beam-on time of 6 h 23 min.

We read the dosimeters the following day. A small loss of dose due to fade is expected, but for the purpose of this study we regard this decrease as negligible. We applied the standard low-dose Cs-137 calibration suggested by Landauer. Initially, we were concerned that the low energy output of the scanner (50 kVp tube potential emits an X-ray spectrum centered roughly in 16 keV–25 keV) would lead to inaccurate readings on the InLights, but since the dosimeters are equipped with filters, the dose equation algorithm in the MicroStar reader can deduce beam energy without a correction factor applied to the 662 keV energy from the original calibration.

The average DDE per scan for all the dosimeters was calculated to be 73.8 nSv. The average SDE per scan was 70.6 nSv, and the average eye-lens dose per scan was 77.9 nSv. The standard deviation (σ) and the coefficient of variation (CV) value of all the dosimeters for the DDE were 0.75 and 0.10 (generally low variance) respectively. For the SDE and lens dose, σ and the CV were 1.26 and 0.16, and 2.08 and 0.29, respectively.

An unexpected aspect of our results is that the measured DDE is higher than the SDE, and this occurrence is worth further examination. The irradiation geometry of the dosimeters could possibly explain this irregularity. It might be productive to conduct further experiments that account for this effect.

The doses we measured are several times higher than those found in the recent AAPM Task Group 217 report [2], but they still equate to only nominal exposure: approximately equal to 24 minutes of natural background radiation and below the recommendation of 250 nSv per screening established by the applicable ANSI/HPS standard [3]. A person would have to undergo approximately 3200 scans per year to exceed the standard's annual exposure limit of 250 µSv/year, a circumstance unlikely even for transportation workers and very frequent fliers.

ROP is Still Dangerous: Breaking Modern Defenses

Nicholas Carlini David Wagner
University of California, Berkeley

Abstract

Return Oriented Programming (ROP) has become the exploitation technique of choice for modern memory-safety vulnerability attacks. Recently, there have been multiple attempts at defenses to prevent ROP attacks. In this paper, we introduce three new attack methods that break many existing ROP defenses. Then we show how to break kBouncer and ROPecker, two recent low-overhead defenses that can be applied to legacy software on existing hardware. We examine several recent ROP attacks seen in the wild and demonstrate that our techniques successfully cloak them so they are not detected by these defenses. Our attacks apply to many CFI-based defenses which we argue are weaker than previously thought. Future defenses will need to take our attacks into account.

1 Introduction

The widespread adoption of DEP, which ensures that all writable pages in memory are non-executable, has largely killed classic code injection attacks. In its place, Return Oriented Programming (ROP) has become the attack technique of choice for nearly all modern exploits of memory-safety vulnerabilities. In a ROP attack, the attacker does not inject new code; instead, the malicious computation is performed by chaining together existing sequences of instructions (called *gadgets*) [27].

In response to this, there has been a large effort to find defenses that protect against ROP attacks. Defenses fall in to two broad categories. The first category of defenses relies on recompilation to remove potential gadgets from the program binary or to enforce the Control-Flow Integrity (CFI) [4] of the binary. The other category of defenses attempts to transparently protect legacy binaries using runtime protections.

In this paper, we present three attack methods that can be combined to break many existing ROP defenses from both of these categories. Our first method breaks the conventional wisdom that it is difficult to mount attacks in a fully *call-preceded* manner, that is, where the instruction before each gadget is a `call`. Many CFI-based defenses rely upon policies similar to this. Next, we show that while most existing ROP attacks consist entirely of *short* gadgets, it is possible to mount attacks which consist of long gadgets as well. Therefore, defenses that distinguish a ROP attack from normal execution by looking for a sequence of short gadgets are not secure. Finally, we examine defenses that record a limited history of the execution state of a process. We show it is possible to effectively clear out any history kept by these defenses, rendering them ineffective.

We use these attacks to break two recent state-of-the-art runtime defenses, kBouncer [23] and ROPecker [11]. These defenses are particularly interesting because they can be deployed on existing hardware, have nearly zero performance overhead, and do not require binary rewriting. kBouncer [23] takes advantage of hardware support for recording indirect branches and examines this history at each system call in order to prevent ROP attacks from issuing any malicious syscalls. ROPecker [11] extends kBouncer in novel ways. In addition to checking for any signs of a ROP attack at each system call, ROPecker additionally checks for attacks at various points throughout program execution.

We show that both of these schemes are broken. While they may detect existing ROP attacks, we give ways of modifying a ROP attack so it will not be detected by either of these defenses. The attacks we develop in breaking these defenses are also applicable to many recent CFI-based approaches, and discuss how our work can be applied to four in particular.

This paper makes three contributions:

1. We introduce three novel ROP attacks methods that demonstrate weaknesses in multiple defenses.

2. We demonstrate these attacks on kBouncer and ROPecker, two state-of-the-art ROP defenses. We modify real-world exploits, which these defenses were shown to prevent, to bypass them.

3. Our attacks provide a baseline set of attacks that can be used to evaluate future ROP defenses.

2 Introduction to ROP Attacks

Return Oriented Programming (ROP) [27] is a generalization of return-into-libc [24] attacks where an attacker causes the program to return to arbitrary points in the program's code. This allows one to perform malicious computation without injecting any new malicious code by only controlling the program's execution flow. It has been shown that ROP can perform Turing-complete computation [30]. We provide a very brief overview of return oriented programming in this section. For a more complete introduction, we refer the reader to [7, 25, 27].

A ROP exploit consists of multiple *gadgets* that are chained together. Each gadget performs some small computation, such as loading a value from memory into a register or adding two registers. In a ROP attack, the attacker finds gadgets within the original program text and causes them to be executed in sequence to perform a task other than what was intended.

Gadget chaining is achieved by influencing indirect jumps executed by the program. Each gadget begins with some useful instructions (e.g., `mov rax, rbx`) and ends with an indirect jump (e.g., `ret` or `jmp *rcx`). The attacker chains gadgets together by controlling the target of a gadget's indirect jump to point to the beginning of the next gadget in the sequence. In a classic ROP attack, gadgets end with the `ret` instruction and the attacker chains gadgets by writing appropriate values over the stack.

Many ROP attacks use *unintended* instruction sequences. Because x86 instructions are variable-width, it is possible that a potentially useful gadget sequence exists when starting at an offset that was not intended to be the beginning of an instruction. Our attacks do not rely on unintended instructions.

In Figure 1, we give an example ROP exploit that adds 0x32400 to the value stored at address 0x4a304120. This exploit begins by initializing two registers. It then reads the value stored at address `eax`, stores it into `eax`, adds `ebx` to `eax`, and stores this value back into memory.

Address Space Layout Randomization (ASLR). One common defense for ROP attacks is ASLR which works by randomly moving the segments of a program (including the text segment) around in memory, preventing the attacker from predicting the address of useful gadgets. Despite ASLR, ROP attacks are still common in the wild for two reasons. First, if even a single module has ASLR disabled, a ROP attack may be formed around only the code in that module. Second, an attacker may use an *information disclosure vulnerability* to de-randomize some module [29].

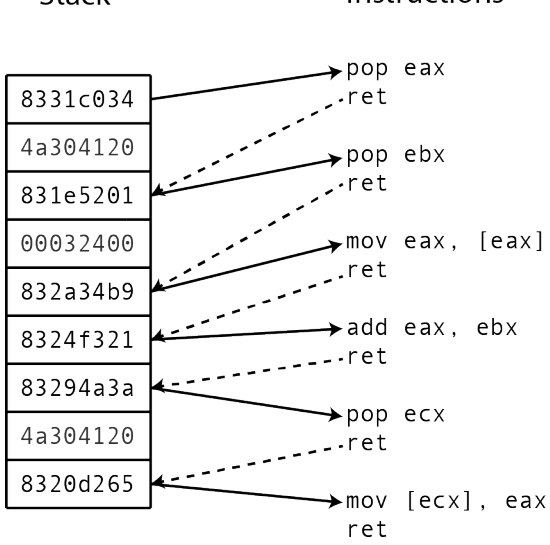


Figure 1: An example ROP exploit which adds the constant 0x32400 to the word at address 0x4a304120. At the left is the stack of the process with the addresses of the gadgets and the values to initialize the registers. At right are the instructions at those addresses.

3 Our Three Attack Primitives

We have identified three building blocks that are useful in attacking ROP defenses:

- *Call-Preceded ROP.* Normally, in a well-structured program, every `ret` instruction returns back to an instruction that immediately follows a corresponding `call`. ROP attacks deviate from this pattern. Therefore, many ROP defenses ensure that every `ret` instruction always targets an instruction that immediately follows some `call`. Our attack demonstrates that this policy is not sufficient: ROP attacks are still possible even when returns are restricted in this way.

- *Evasion Attacks.* It is common for defenses that monitor program execution at runtime to have a method of classifying execution as either "normal execution" or "gadget". Evasion attacks involve using gadgets that the defense classifies as "normal."

- *History Flushing.* Some defenses maintain only a limited amount of history about execution and inspect this history periodically. We can bypass defenses with this property by flushing the true history (cleansing the history of all signs of the ROP attack) and then presenting a new, fake view of history that the defense will not classify as an attack.

Each of these three attack primitives bypasses a common defense mechanism. This section gives more detail about each of these three primitives. We then combine them in different ways to mount our full attacks on kBouncer [23] and ROPecker [11] in the following sections.

3.1 Call-Preceded ROP

The call-preceded policy. We say that an instruction is *call-preceded* if the instruction immediately preceding it is a `call` instruction. Many ROP defenses [6, 23, 32, 34] apply the following policy: any time a `ret` instruction is executed, its target must be a call-preceded instruction.

This policy seems helpful for defending against ROP attacks. In well-structured programs, calls and returns usually come in pairs. Any address that is returned to was almost always pushed by a `call` instruction previously. In a ROP attack, gadgets use the `ret` instruction to chain gadgets together, so this policy dramatically limits the space of candidate addresses where gadgets can be chosen from. For instance, one evaluation found that only 6% of gadgets are call-preceded [23]. Thus, one might intuitively expect the call-preceded policy to significantly increase the difficulty of mounting a ROP attack.

Using only call-preceded gadgets. Despite this intuition, we find that it is possible to mount ROP attacks in a fully call-preceded manner, where all gadgets start at a call-preceded address. The key idea is we allow gadgets to be more complex. This increases the space of candidate gadgets enough to find many call-preceded gadgets. By allowing our gadgets to be long and contain direct jumps or even conditional jumps, we find many more useful gadgets. In our experiments (see § 8.2), 70KB of binary code was sufficient to mount fully call-preceded ROP attacks.

3.2 Evasion Attacks

Classification-based defenses. Other ROP defenses work by monitoring the runtime behavior of a process and try to detect ROP attacks by classifying segments of execution as either "gadget" or "non-gadget", using some signature that is intended to characterize attributes of ROP gadgets. One of the most common approaches used to classify execution, as used in [11, 23], uses a length-based classifier. Existing ROP attacks tend to consist of long sequences of short gadgets, and so these defenses use this as their heuristic to classify gadgets.

These defenses separate the execution trace into segments of ordinary instructions, separated by indirect instructions (e.g., returns, indirect jumps). A length-based defense classifies each segment as gadget or non-gadget by examining its length: a short segment is classified as a gadget and a long segment as a non-gadget. If the defense observes too many short segments within some window, it reports a ROP attack.

Using gadgets that look like benign execution. A powerful attack on such defenses is to look for instruction sequences that would be classified by the defense as a non-gadget, but that perform some useful computation. These can then be used as stealthy gadgets in a ROP attack.

Length-based classifiers are particularly easy to evade. A simple attack is to use long gadgets, since these will be incorrectly classified by the defense as non-gadget. We demonstrate that it is possible to mount a ROP attack that contains a mixture of both short and long gadgets, thus evading many published detectors.

More generally, one could imagine future ROP defenses that rely on other heuristics for distinguishing ROP attacks from normal program execution. An *evasion attack* is one that will be classified by the defense as normal, but in reality allows the attacker to mount a ROP attack.

3.3 History Flushing

History inspection defenses. There are many runtime defenses that inspect program execution at different points throughout its execution. Typically, these defenses keep only a limited amount of history about the program's execution, and so must decide whether an attack is occurring or not based upon information saved in the recent past. Usually, performance considerations rule out constantly monitoring all execution, so this inspection process is only invoked at certain points (e.g., when the application issues a system call).

Using gadgets to hide history. Such defenses can be fooled by preventing them from seeing any evidence of a ROP attack. We perform the ROP attack when they are not watching, periodically performing enough innocuous actions to wipe the history clean of any evidence of the past ROP attack before the defender's inspection process is invoked. While the defender is running, we do not attempt to make progress towards our attack goal. Instead, we insert effective no-op instructions so that the defender does not see any evidence of attack.

Though similar, this attack is different from an evasion attack. An evasion attack attempts to make progress in the attack while being *continuously* monitored by the defender. In a history flushing attack, there is a period of time when the defender is not running, when we make forward progress. Before the defender runs, we clear out this history so it is not visible to the defender, but do not attempt to make forward progress while the defender is watching. After the defender has made its observation, we continue with our attack.

For instance, kBouncer uses the Last Branch Record, a

hardware feature that records the 16 most recent indirect jumps. Our history-flushing attack on kBouncer performs the bulk of the ROP attack, then performs 16 innocuous indirect jumps to remove the evidence of the ROP attack from the Last Branch Record. As we show (§ 8.3), this prevents kBouncer from detecting the ROP attack.

4 Attack Goal & Threat Model

Attack Goal. The goal of each of our attacks, without loss of generality, is to issue a single syscall. It is usually enough to issue a `mprotect` (on Linux) or `VirtualProtect` (on Windows) system call to make a page in memory both writable and executable; after that, exploitation is trivial.[1]

This is not the only possible goal an attacker may have. There are other methods of attack that do not involve issuing system calls [10]. We do not consider them in this work, although our results suggest these attacks are equally possible, and in some cases even trivial.

Threat Model. At a minimum, we assume that an attacker has a known exploit that allows control of the instruction pointer in the future. A stack overflow is sufficient; a heap overflow that allows an arbitrary memory write to a function pointer is also sufficient; as is directly overwriting other function pointers. We assume the attacker knows that the defense is present and knows how it works. We assume that DEP is enabled, so no page is both writable and executable. We focus on the case where the program contains at least one library whose executable region has not been randomized with ASLR, or where all modules have ASLR enabled but there exists a memory disclosure vulnerability, as this is the situation that modern ROP attacks typically exploit.

We also assume that there exists some way of running arbitrary code if the new defenses were not present. We do not claim to create attacks that allow running arbitrary code in all situations; we only hope to show that if it is possible to mount a ROP attack when the defense is not present, then it is possible when it is present.

5 Defeating kBouncer

5.1 Overview of kBouncer

Pappas et al. introduced kBouncer [23], a scheme that uses indirect branch tracing to detect ROP attacks. At a high level, kBouncer periodically pauses execution of the program, inspects recent execution history, and then either allows the process to proceed or kills it.

[1] Alternatively, if we can execute the `execve` syscall, we can spawn a second process running an arbitrary program.

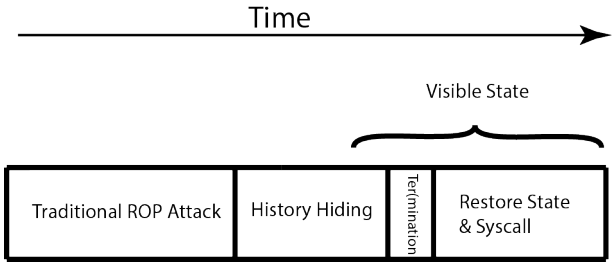


Figure 2: Overview of our history hiding attack on kBouncer. We mount a traditional ROP attack, insert a number of innocuous gadgets to hide this from kBouncer, and finally restore registers and issue the desired syscall.

kBouncer uses the Last Branch Record (LBR), a feature of modern Intel CPUs, to inspect the last 16 indirect branches taken each time the program invokes a system call. kBouncer checks two properties of the history stored in the LBR. First, it verifies that all `ret` instructions in the LBR returned to a call-preceded address. Second, if the eight most recent indirect branches are all gadget-like, the process is killed. kBouncer defines a sequence of instructions as *gadget-like* if there exists a flow of execution from the first instruction executed to any indirect branch in under 20 instructions.[2] kBouncer is very efficient: it only needs to check the LBR during system calls and only checks 16 different entries in the LBR.

5.2 History Hiding Attack

5.2.1 Attack Overview

We dub our first attack on kBouncer the *history hiding attack* (see Fig. 2). At the core of kBouncer is the assumption that an attack can be detected by inspecting the state of the process at the syscall interface, after the attacker has already gained control of the system for a potentially unbounded period of time. After mounting a traditional ROP attack to prepare the state of memory (and possibly defeat ASLR, if required), we use a history flushing attack to clear evidence of the attack from the LBR. Finally, we use an evasion attack and a few carefully-chosen gadgets to issue the syscall.

We call a process state *valid* if kBouncer's inspection method will not detect an attack when run from that state. A state is valid if all of the entries in the LBR whose source is a `ret` instruction have a call-preceded destination, and if at least one of the last eight entries has more

[2] kBouncer cannot observe the actual path of execution taken during a sequence of instructions between two indirect jumps, so it cannot count the number of instructions actually executed between two indirect jumps. It can only observe the beginning and end of that sequence. For this reason, kBouncer conservatively treats a sequence as gadget-like if it starts with an instruction that can reach an indirect jump in less than 20 instructions.

than 20 instructions between source and the nearest indirect branch. We show that it is easy to return to a valid state while simultaneously maintaining control of the process. The steps of the history hiding attack are as follows:

Initial exploitation. Initially, we mount a traditional ROP attack in whichever way is easiest. We ignore the fact that kBouncer is running and use any gadgets we would like, call-preceded or not. We then prepare memory so we are ready to make the syscall, but we do not invoke it yet.

Hide the history. At this point in our exploit, we are ready to make the syscall, but if we were to actually issue it, kBouncer would detect an attack. To fix this, we must bring the process into a valid state without losing our progress from the prior step. To do this we use the history-flushing primitive discussed previously. As a side effect of using the flushing primitive, the registers may be clobbered, but important memory locations will remain unchanged.

Restore registers and issue the system call. After bringing the process into a valid state, we restore the registers to their desired values while maintaining a valid state. Then, we issue the system call. This is via an evasion attack: because the task is relatively simple, it can be accomplished with fewer than 8 call-preceded gadgets.

```
                    add  [esp+17Ch],ebx
                    mov  ebx,[esp+17Ch]
(1)                 sub  ebx,ebp
   pop  ebp         jmp  A
   ret              ...
                A:  add  [esp+64h],ebx
(2)                 jmp  B
   jmp  A           ...
   ...              mov  esi,[esp+1C0h]
A: mov  eax,3       lea  eax,[esi*8-4]
   ret              sub  eax,[esp+64]
                    and  eax,7h
(3)                 mov  edi,[esp+64]
   cmp  eax,6       lea  eax,[edi+eax+4]
   jbe  B           shr  eax,3
   ...              cmp  eax,esi
B: ret              jbe  C
                    ...
(4)             C:  mov  eax,[esp+1C0h]
   xor  eax,eax     add  esp,19Ch
   ret              pop  ebx
                    pop  esi
(5)                 pop  edi
   mov  [eax],0     pop  ebp
   ret              ret
```

(a) Flushing Gadgets (b) Termination Gadget

Figure 3: Examples of the two types of gadgets used by our history-hiding attack on kBouncer. A flushing gadget flushes the contents of the LBR. A termination gadget brings the system into a valid state.

5.2.2 Initial Exploitation

This step prepares memory to make it as easy as possible to issue the syscall in as few gadgets as possible after the history has been flushed. In particular, we prepare all of the arguments for the system call and save them in some easily recoverable location. We make no restrictions on the methods the attacker may use during this step of our attack. Because we are going to hide our history, kBouncer will not observe anything performed in this step. Since ROP gadgets are Turing-complete, we are able to perform arbitrary computation during this phase, so this step is straightforward to implement.

5.2.3 Hiding the History

Hiding history through LBR flushing. We use a history-flushing primitive, built from two gadgets (Fig. 3), to remove all traces of our attack from the LBR:

1. A short *flushing* gadget: a simple call-preceded gadget that performs a `ret`, and ideally does not modify many registers.

2. A long *termination* gadget: a call-preceded gadget that is long enough for kBouncer to not classify it as a gadget: there must be at least 20 instructions along every possible control path from the start of this gadget to any indirect branch.

We use these two gadgets as follows. First, we repeatedly use the flushing gadget to completely clear the contents of the LBR until it only contains the flushing gadget repeated 16 times. Though the LBR has been flushed and contains no history of the previous ROP attack, the state is still not valid. If kBouncer were to be invoked at this point, every entry in the LBR would be classified as a gadget by kBouncer and an attack would be detected.

We now use the termination gadget. The purpose of this gadget is to bring the LBR into a valid state by making at least one of the last eight entries in the LBR have length greater than 20. That is, the termination gadget is used to *terminate* kBouncer's backwards search for gadget-like sequences. We make no assumptions about the register state after the termination gadget is executed: the only requirement is that after we use it, we still have control of instruction flow.

```
4a833dd4        dec     ecx
4a833dd5        fmul    [4A88BBC8h]
4a833ddb        jne     4A833DD4
```

Figure 4: An example of a context switch gadget found in `icucnv36.dll`.

Note that during the first step where the attacker prepares memory, the attacker may perform arbitrarily complex calculations. This may make it possible to initialize registers and memory so that executing the flushing gadgets and then the termination gadget results in exactly the desired state to issue the syscall. However, this is not always possible. For example, the termination gadget may set `eax` to 0, but issuing the syscall may require `eax` to be 7. Our attack handles this situation by later restoring register state (described below).

Because the termination gadget is over twenty instructions long and might contain conditional branches, it is sometimes necessary to initialize registers and memory to meet the preconditions for successful execution of the termination gadget. First, we need to ensure that any conditional branches in the termination gadget will be followed in a specific manner. Second, memory reads and writes must not fault and crash the process. This is often as easy as initializing registers to specific values before using the termination gadget. We have found that termination gadgets are very common, and that it is often easy to find termination gadgets that perform only a few conditional branches and memory reads and writes (see § 8.3.1).

History hiding by itself does not defeat kBouncer, but it simplifies the attacker's job from expressing the entire attack using call-preceded gadgets to expressing only the final step of the attack using call-preceded gadgets.

Hiding history through context switching. We also found an alternative way to flush history. The LBR is shared across all user-space processes. This lets us flush the LBR using a single gadget, the *context switch* gadget. A context switch gadget is one that will run for many seconds and will not contain any indirect branches. The simplest way to find such a gadget is to look for loops that perform a very limited computation using only registers, see Fig. 4 for one such example.

To flush the LBR, we call the context switch gadget once. Due to the number of cycles this gadget takes to execute, it is almost certain that there will be several context switches to other user threads during its execution. When this happens, the other thread will write its own entries to the LBR, flushing all history of our prior attack. Eventually, when our context switch gadget finishes, the LBR will be in a valid state as long as the other process was not under attack, as the LBR is now full of innocuous entries from the other process.

Future hardware could save and restore the LBR on context switches, which would prevent this method of history flushing. Therefore, we did not use this approach in our case studies (§ 8); instead, we used flushing and termination gadgets, which would suffice to hide history even if the LBR was saved and restored on each context switch.

5.2.4 Restoring Registers with Returns

We must now restore the registers to their desired values in order for the syscall to proceed. This is by far the simplest step and can be usually be accomplished with a few gadgets that pop register values off the stack. kBouncer will be able to observe each gadget we use, so each one must be call-preceded and we must use fewer than eight.

This step is often very easy because of the x86 calling convention: the procedure being called must restore almost all of the registers, so procedures tend to begin by pushing all of the registers onto the stack and end by popping those values off to restore them. This allows us to find a gadget that pops all the registers off the stack and then returns. Usually, we can find all the (call-preceded) gadgets we need in this way.

5.2.5 Restoring Registers without Returns

There are other ways to restore register state. We now discuss four alternative methods. The first two are existing techniques that can be applied here, but in our experience are difficult to apply in practice due to the fact that we must use fewer than eight gadgets. We have found the later two techniques more applicable in practice.

ROP without return instructions. Checkoway et al. found it is possible to mount a ROP attack by looking for a `pop` followed by an indirect jump (e.g., `pop edx; jmp *edx`) [8]. This instruction sequence is functionally identical to a `ret`, and so can simply be used in its place. However, these sequences are less common.

Jump Oriented Programming (JOP). JOP attacks use register-indirect jumps to chain gadgets together. Unfortunately, each useful gadget must be followed by a dispatcher gadget, which is used for chaining. Since we must restore register state with at most eight gadgets, if we want to use JOP, we are limited to four useful JOP gadgets.

Using Non-Call-Preceded Gadgets. Occasionally, it may be easier to use non-call-preceded gadgets. We can invoke a non-call-preceded gadget using a *reflector* gadget. A reflector gadget is a call-preceded gadget that ends in a register-indirect jump; it can be used to jump to any gadget we like, call-preceded or not. This is because kBouncer imposes no constraints on indirect jumps. Our

experience is that this trick is rarely needed in practice, but sometimes it makes constructing the attack easier.

Call Oriented Programming (COP). We have found an alternate method of mounting a ROP-like attacks without using `ret` instructions. We call our approach Call-Oriented Programming (COP). Instead of using gadgets that end in returns, we use gadgets that end with indirect calls. This may at first seem trivially similar to jump-oriented programming, but there is one important distinction: indirect calls are usually memory-indirect (the location to which control is transferred is determined by a value in memory, not directly by the value of a register). As a result, COP attacks do not require a dispatcher gadget. In a COP attack, gadgets are chained together by pointing the memory-indirect locations to the next gadget in sequence. The initialization of these memory locations can be done *in advance*.

This allows our attack to set up these memory locations before the history hiding, then restore register state using COP gadgets. As long as fewer than eight COP gadgets are used, kBouncer will detect no attack. When mounting a COP attack, it is trivial to directly issue the desired system call as well: the final gadget in the sequence will point to the system call to be issued.

We have found that memory-indirect calls, and in particular COP gadgets, are common. They are even more common than call-preceded gadgets that end in a `ret`. There are two reasons why this is the case. First, with dynamically linked libraries, all calls to functions outside of the current module are indirect calls, because the function location is not known in advance. Second, most object-oriented code relies on memory-indirect calls (e.g., the vtable in C++).

COP attacks do not eliminate the need for `ret`-based gadgets. Initializing a COP attack is much more difficult: the attacker must have control of program flow, must overwrite specific indirect-call locations, and must control the stack. This usually is not possible with a single exploit. Therefore, it is natural to combine a ROP attack (for initial setup) with a COP attack (for restoring registers).

5.2.6 Issuing the System Call

The final step of our attack is to issue the desired syscall. We usually accomplish this by calling the appropriate `libc` or `kernel32` wrapper function.

There is one complication. We cannot simply return directly to the beginning of the desired function (e.g., `mprotect`, `VirtualProtect`) as a normal ROP attack would. When kBouncer is in place, this is not possible: the attack would fail because the start of this function is not call-preceded. We have found three different ways to call a function without directly returning to it.

```
    call [7C37A094]
A:  mov eax,[_osplatform]
    jmp B
    ...
B:  dec eax
    neg eax
    sbb eax,eax
    and eax,103
    lea ecx,[ebp-0Ch]
    push ecx
    inc eax
    push eax
    push [EBP-8]
    push [EBP-4]
    call [VirtualProtect]
```

Figure 5: A call-preceded call to `VirtualProtect` in `msvcr71.dll`. The attacker can return directly to `A`.

1. We can use a *reflector gadget*: a call-preceded gadget that ends with a register-indirect jump. This allows us to simply set a register to point to the function we wish to call and then return to the reflector gadget. This is the simplest approach if a reflector gadget can be found.

2. It is still possible to exploit the desired function even if no reflector gadgets are available. This is achieved by finding an call to the desired function somewhere in the program's code and looking backwards in the instruction sequence for a preceding `call`. Fig. 5 shows an example where the `msvcr71.dll` binary directly calls `VirtualProtect`.

3. It is sometimes possible to return into the middle of a desired function, right after a `call` instruction. For example, `execv()` launches a shell with a string and an array of arguments (Fig. 6). If we initially initialize `rax` to contain a valid environment pointer, we can call `execv` by returning directly to `<execv+18>`, which is call-preceded.

Any of these can be used to complete our attack.

5.3 Evasion Attack

Our history hiding attack breaks kBouncer by taking advantage of its limited history. If kBouncer were extended to have a complete view of history, would it become more effective? We show that, even if the LBR were of infinite size, kBouncer could still be broken by an evasion attack.

Our attack is similar to the history hiding attack (§ 5.2), except that the initial preparation phase is mounted using only call-preceded gadgets. This eliminates the need

```
<excve>:
    push   rbp
    mov    rbp,rsp
    push   r14
    push   rbx
    mov    r14,rsi
    mov    rbx,rdi
    call   _NSGetEnviron
    mov    rdx,[rax]
    mov    rdi,rbx
    mov    rsi,r14
    call   execve
    mov    eax,-1
    pop    rbx
    pop    r14
    pop    rbp
    ret
```

Figure 6: Disassembly of the `execv` function in libc on our system. The call to `_NSGetEnviron` allows a call-preceded return directly into this function.

for a flushing gadget, the only piece that an infinite-LBR kBouncer would preclude. Therefore, our attack consists of a (call-preceded) setup, a (call-preceded) termination gadget, followed by (call-preceded) register restoration and syscall.

This yields a successful evasion attack on kBouncer. By using only call-preceded gadgets and by breaking up the chain of short gadgets with a long termination gadget, kBouncer can see the entire attack but still will not recognize it as an attack. Our experiments show that if over 70KB of program text is available, then there are enough call-preceded gadgets that this attack is possible (see § 8.2).

6 Defeating ROPecker

6.1 Overview of ROPecker

ROPecker [11] is a ROP defense that builds on ideas found in kBouncer. ROPecker differs from kBouncer by running its inspection method more frequently and inspecting the program state more thoroughly at the time of inspection. The actual policy it enforces is very similar to the kBouncer policy.

In ROPecker, only a few pages are ever marked executable at one time. We call these pages the *executable set*. Whenever a page not in the executable set is executed, a page fault is generated and ROPecker pauses process execution to check for an attack. If ROPecker does not detect an attack, it marks the new page as executable, marks the least recently executed page as non-executable, and resumes the process. ROPecker also runs its detector whenever the process invokes a syscall as kBouncer does.

ROPecker's detector is more sophisticated than kBouncer's in that it looks at both the recent past and projects forward into the near future. Similar to kBouncer, ROPecker classifies the current state as an attack if there is a long chain of *gadget-like* sequences in the LBR (the recent past). In addition, ROPecker attempts to emulate what will happen in the near future once the process is resumed. It counts the number of gadget-like sequences that are about to execute. If the sum of the number of gadgets found in the LBR and the number of gadgets looking forward exceeds some threshold, ROPecker classifies this as an attack.

ROPecker's emulation works by disassembling the instruction stream from the instruction that is about to execute when the page fault occurs. If there is a short sequence of instructions that leads to an indirect jump, ROPecker classifies this as a potential gadget. ROPecker will then emulate the effects of each of the instructions leading to the indirect jump in order to compute where this jump will go. ROPecker follows this indirect jump and starts disassembling again. When it reaches an instruction where there is not a short sequence of instructions leading to an indirect jump, it stops the search. ROPecker then counts the number of indirect jumps followed, and classifies each of those as gadgets.

ROPecker verifies that from the current execution point there are not 11 gadget-like sequences of instructions.[3] ROPecker classifies an instruction sequence as a gadget if it contains six or fewer instructions ending in an indirect branch, with no direct or conditional branches along the way.

6.2 The Repeated History Hiding Attack

6.2.1 Attack Overview

We show how to break ROPecker using a *repeated history hiding* attack. This attack repeatedly invokes the history-hiding primitive, introduced in § 3.3, just before ROPecker's detector is about to execute. We again define a state to be *valid* if the inspection method will not detect an attack. The state must be valid at two points in time: whenever a new page is loaded in to the executable set and whenever a syscall is executed.

Our attack alternates between three phases, as depicted in Fig. 7. The *loading phase* loads useful pages into the executable set. The *attack phase* invokes gadgets on these pages. The *flushing phase* mounts the history hiding attack from § 5.2 using only gadgets from the pages that are

[3] The ROPecker paper does not pick a specific parameter for the maximum number of gadgets that may execute consecutively. It suggests this number is chosen between 11 and 16, so we conservatively pick 11. Our attacks are made easier if a larger number is chosen.

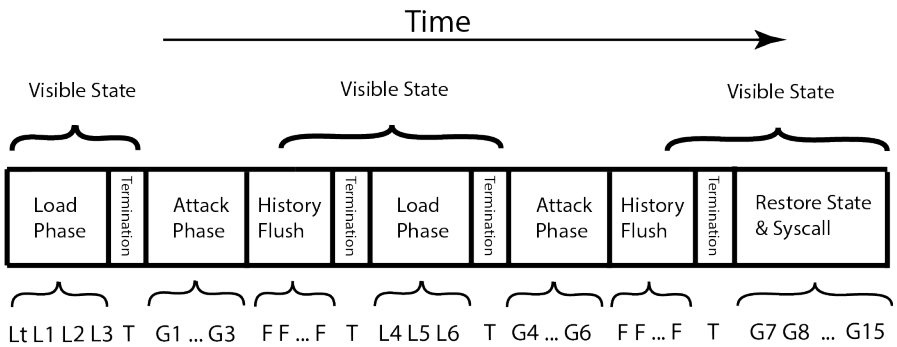


Figure 7: An overview of the repeated history hiding attack on ROPecker. `Ln` gadgets load page `n`. `Lt` loads the termination gadget. `Gn` invokes a gadget on page `n`. `F` is a short flushing gadget, and `T` is a long termination gadget.

in the executable set. We may need to execute each of these three phases multiple times to achieve our goal. We conclude with one final step which actually issues the desired syscall after restoring the required state. Because we use only gadgets in the executable set during each attack phase, ROPecker's detector will execute only when new pages are loaded, which allows us to reason about what will be visible to ROPecker.

6.2.2 Attack Phases

Initialization. Prior to our attack, we insert a termination gadget, which will stop ROPecker from looking further back in the LBR. This long termination gadget is identical to the one used in the kBouncer attacks. This ensures that when ROPecker next runs, it will not count any functions on the call stack prior to initialization as gadgets.

Loading Phase. We load useful pages into the executable set by invoking a page load gadget on each page we want added to the executable set. A *page load gadget* is any call-preceded gadget on that page, which has two properties: first, it must leave the attacker with control of the instruction flow; and second, it must not crash the process. These two requirements are not difficult to meet: any useful gadget is also a page load gadget. The ROPecker detector will run immediately before each page load gadget is invoked. After invoking each set of page load gadgets we call the termination gadget to prevent the detector from looking forward any farther into the future.

ROPecker will not detect an attack because each sequence of page load gadgets is immediately preceded and followed by a termination gadget. When a page fault occurs, ROPecker will count the number of visible gadgets looking backwards in the LBR and forwards as far as it can see. Looking backwards will stop at preceding termination gadget, and looking forward will stop at the subsequent termination gadget. Thus, ROPecker will count the number of page load gadgets. By limiting the number of consecutive page load gadgets, the attacker can evade detection during this phase.

Attack Phase. Now that the useful pages have been loaded, we can use any gadgets on these pages to mount an attack, ignoring any defense which may be running. As long as we use only gadgets on these pages, the defense will never trigger.

Recall that these three phases are repeatedly executed, so no one attack phase needs to perform the entire attack. Instead, the attack can be distributed among multiple attack phases, making each one simpler.

History Hiding. After invoking gadgets on these pages, we now use the history flushing primitive before the detection method next runs. We use the same method we applied against kBouncer to clear the LBR. In particular, we invoke a short flushing gadget enough times to fill the LBR with innocuous entries, then invoke the long termination gadget (which was loaded previously). When the ROPecker detector next runs, it will see no attack prior to this point in time.

6.2.3 Segmenting the Attack Payload

When mounting this attack, we must carefully pick which tasks to perform during each attack step. Because the flushing and termination gadgets clobber some register state between each attack step, it is important to pick small independent operations for each step of the attack.

For any given attack, it may not be possible to modify it to work as an attack which bypasses ROPecker. Instead, attacks must be formed with ROPecker in mind. Each step in the attack must be constructed to use only a limited number of gadgets, so that its work can be saved before loading in a new set of gadgets.

Often, we start by computing the address of the desired `libc` function we wish to call (e.g., `mprotect`) either by adding a constant to the address of some other function

in `libc`, or by loading it directly. We store the result in memory. In the next attack step, we compute the address of the page we wish to mark as executable (typically on the stack). We continue in this way, computing any other needed constants in separate attack steps. We then restore register values and call `mprotect` on the desired page. Finally we can execute a traditional payload with data we have written to this page.

6.2.4 Selecting Pages to Load

Since the executable set can contain only a few pages at one time, we must choose these pages with care. The naive approach is to select each page to load for one useful gadget on that page, and call each gadget exactly once. We have found that this simple method works well in practice in most cases. Because the flushing and termination gadgets may clobber a few registers, we may need reserve one or two of those gadgets to load and save registers to memory, so that a task can be partially completed in one attack step.

A more advanced method is to pick pages that contain multiple gadgets. In our evaluation, we found that in practice there tend to be many "useful" gadgets on the average page. Thus, by selecting the pages carefully, we can find pages with enough useful gadgets. This is enough that we can attack ROPecker even when the size of its executable set is limited to just one or two pages.

6.2.5 Issuing the Syscall

Once we have executed sufficient load/attack phases to set up the state of the process, we append one final step to actually issue the desired syscall. This step is not executed multiple times: it is done only once at the very end.

During this step, we flush history, invoke the termination gadget, and then issue the syscall using one of the three methods from § 5.2.6. We perform this step using at most 10 gadget invocations so that ROPecker will not detect an attack when it examines the LBR at the syscall.

Conveniently, it is possible to use any gadget in the entire binary during this step, even if it is not contained within the executable set. No page loading gadgets are needed. This works because there will be at most 10 gadgets between the termination gadget and the syscall. Thus, even though ROPecker's detector may run during this step (if we use a gadget that's not in the executable set), its count of the number of gadgets will be below 11, the threshold for detecting an attack.

Note that, in particular, an attack which requires fewer than ten gadgets to execute can skip the load/attack phases and directly issue the syscall in this way.

6.3 The Evasion Attack

We now present the ROPecker evasion attack, an alternate attack that would break ROPecker even if the size of the executable set were reduced to just *one* page. As a side benefit, in our experience the evasion attack makes it easier to automate attacks in practice than the repeated history hiding attack of § 6.2.

At a high level, the idea is that we will let ROPecker inspect the execution of our attack at arbitrary points in time. We ensure that no matter when its detector runs, it will never detect an attack. We achieve this through an evasion attack similar to the one presented on kBouncer (§ 5.3).

The ROPecker evasion attack works by inserting a termination gadget in between every ten useful gadgets. When the detector runs, it will check forward and backward to count the number of gadgets in use; there will be fewer than 11 gadgets, the threshold for detection, so ROPecker will not detect the attack.

The authors of ROPecker note that this attack may be possible in § VII(b) of their paper [11]. They propose a mitigation for such an attack. We show that even their mitigation is broken.

The ROPecker mitigation. ROPecker detects an attack if there are more than ten consecutive gadgets. The extended version of ROPecker records how many gadgets existed in previous runs of the detector. It detects an attack if the number of gadgets which executed in the last T runs is larger than some threshold. While it is possible for there to be 10 sequential gadget-like returns in benign program execution, it is unlikely for there to be 10 sequential gadget-like returns *T times in a row*.

Conceptually, this is analogous to running the detection mechanism both forwards and backwards, allowing up to $T-1$ long gadgets before stopping the search. An attack is detected if the number of gadgets found by this extended search is greater than some threshold.

This defense *does not* help against our repeated history hiding attack. In that attack, ROPecker only ever sees as many gadgets as pages that are being loaded. This constant is usually very small (e.g., two or four). The ROPecker authors observed that benign execution does occasionally execute four sequential gadget-like chains (with frequency 0.58%). This frequency is large enough that signaling an attack if there are four gadgets repeated three times would cause too many false positives.

Breaking the mitigation. The extended version of ROPecker can be broken by a simple modification of our evasion attack: instead of invoking the termination gadget once, invoke it T times in a row. We alternate making one step of useful progress (with ten useful gadgets) with invoking the termination gadget T times. This prevents

ROPecker from detecting consecutive long chains of gadgets. Instead, it sees a long chain followed by several short chains, which will not trigger the defense.

Practicality. One might wonder whether evasion attacks are practical. If, between every ten useful operations, we must potentially destroy our progress, can we achieve any useful computation?

We found it is still possible to perform useful tasks even when inserting a termination gadget (or, potentially multiple termination gadgets) in between every ten useful gadget (see § 8.3). We save register state to memory before each termination gadget and restore it afterwards. It is only necessary to save and restore registers that are both clobbered by the termination gadget and used by the rest of our attack. In our experience, it is often possible to find termination gadgets that only clobber one or two registers. This allows for many gadgets that make forward progress, with a few dedicated to saving and restoring state.

6.4 Attack Comparison

These two attacks are useful in different circumstances. The most important difference is when the detection mechanism runs. In repeated history hiding, the detection only ever runs after a history flush, and so the defender can never even see what the attacker is doing. In the evasion attack, the defender is continuously monitoring the attack progress. This leads to the key distinction between the two attacks. In repeated history hiding, we have a very limited set of gadgets, but may use them an unbounded number of times before flushing. In the evasion attack, we have all of the gadgets in the program available to us, but must flush every ten gadgets.

7 Fixable Attacks on ROPecker

We now discuss several ways in which ROPecker is broken that our attack does not rely on. That is, the attacks discussed in the previous sections work *even if we improve ROPecker's detection mechanisms* to prevent each of the following specific attacks. We believe these modifications are possible, and it is only the engineering difficulties of obtaining a low overhead that explains why they are not currently implemented. Because of this, we do not base our previous attack on these fixable implementation issues.

Gadget definition does not allow any branches. ROPecker's definition of a gadget is overly specific and does not allow gadgets to contain either direct or conditional branches. In comparison, we have found that kBouncer's definition of a gadget is strong: it is difficult to find gadgets of length twenty or more that perform useful computation.

ROPecker's choice to not follow any direct or conditional branches is a flaw that, while allowing for a more efficient implementation, makes exploitation nearly trivial. This decision allows an attacker to flush the LBR, and to stop the forward-inspection algorithm, with a no-op-like gadget that jumps directly to a return instruction. This form of gadget is pervasive in program binaries and allows for a much simpler termination gadget that does not clobber any register state.

In fact, when evaluating the practicality of our attacks on kBouncer *before becoming aware of ROPecker*, nearly all of our exploits contained at least one *useful* gadget that would not be classified as a gadget by ROPecker's definition.

Gadget chain threshold is too short. ROPecker's choice to define gadgets as being a sequence of six or fewer instructions makes it nearly trivial to find gadgets that have a predictable behavior while still being classified as a non-gadget by ROPecker. For example, on 64-bit systems, the gadget consisting of popping off registers `r10` through `r15` followed by a `ret` is seven instructions long: not only is this a useful gadget, it is very common. ROPecker's failure to recognize it as a gadget is a serious limitation of ROPecker.

The set of risky system calls is not complete. ROPecker's set of *risky system calls* is too limited and needs to be updated to more closely match those used in kBouncer. Because ROPecker is designed for Linux and kBouncer for Windows, we cannot simply replace one with the other. However, other than performance reasons, there is no reason to not defend all system calls.

8 Evaluation

The attacks discussed in the previous sections are practical. We evaluate these attacks by modify real-world exploits, as well as by demonstrating that only 70KB of code is needed to mount purely call-preceded attacks.

8.1 Our Tool

We built a tool to assist our efforts in finding attacks on real-world exploits. It does not automatically break either of these two defenses, but assists in finding useful gadgets. We wrote our tool as a 1K line Python program. It takes as input a disassembled object file (from `objdump`), and therefore only inspects *intended* instruction sequences: even though there may be unintended instruction sequences which are call-preceded, we ignore these.[4]

[4] Even though ROPecker does not enforce gadgets are call-preceded, we still use this tool to evaluate ROPecker, as we find it is sufficient to identify useful sequences.

Binary	Setup	Flush	Syscall
diff	8	3	2
grops	4	3	4
lsof	12	2	3*
ltrace	4	2	2
grub-mkimage	4	4	3
strace	17	4	2*
pic	11	2	3
apt-get	14	3	2*
info	13	3	3*
apt-ftparchive	4	3	2

Table 1: The number of gadgets for the three steps in our kBouncer attack for binaries from `/usr/bin/`. Entries marked with an asterisk have success probability of $\geq 99.99\%$, the rest with 100%.

Our tool first enumerates all potential call-preceded gadgets. We implemented a simple symbolic execution framework to determine the effects of each of these potential gadgets. This system is not complete, but it models some of the effects of many common instructions.[5] It computes and outputs the path constraints that must hold to follow the conditional branches in a gadget. It also outputs the list of modified memory locations, accessed memory locations, and the new values of updated registers at the end of execution.

The tool returns a list of gadgets sorted by ease of use: gadgets with fewer conditional branches and fewer memory locations which must be valid rise to the top. Each gadget is marked with a hint on how it might be useful (e.g., that the gadget is a memory-load gadget, or that it computes the sum of two registers). It also provides us with a list of termination gadgets, sorted by ease of use and the number of other registers they clobber.

8.2 Fully Call-Preceded Attacks

How practical are fully call-preceded ROP attacks? Our measurements indicate that they are quite practical. The Q ROP compiler [26] is able to mount a ROP attack in 80% of binaries over 20KB in size. Given that only 6% of gadgets Q finds are call-preceded, we would expect that with 333KB of binary, we could achieve similar results. We actually found that it is possible to exploit 10 out of 10 programs we analyzed of size 70KB or larger.

We analyzed 10 binaries from `/usr/bin` on Ubuntu 12.04. In particular, we selected the first 10 binaries that have ASLR disabled and have more than 20k instructions (70KB binary size). In all 10 cases, we were able to find enough gadgets to mount a fully call-preceded history hiding ROP attack on kBouncer. Table 1 shows, for each of these ten binaries, the number of gadgets used for in each of the three phases of our ROP attack. Attacks marked with an asterisk have a success probability of $\geq 99.99\%$ due to the possibility of a module crossing a 32-bit boundary. All other attacks have a 100% success probability.

In each of these binaries, we use only the code present in the actual binary, not any other linked libraries. We are not arguing that these binaries are vulnerable to attack; we are only attempting to determine how much program text is required to mount fully call-preceded attacks.

We believe there to be two main reasons why we were so successful. First, we manually analyzed these binaries in order to construct a ROP attack, whereas Q is an automated tool. However, given Q's sophisticated analysis, we do not believe this accounts for all of the difference. We suspect that even though only 6% of gadgets are call-preceded, they have more diversity and thus are disproportionately likely to cover the space of different kinds of gadgets that are needed.

8.3 Modifying Real-World Exploits

We now evaluate the difficulty of modifying real-world exploits to bypass both kBouncer and ROPecker. To choose our exploits, we pick the ROP attacks that were shown to be prevented by kBouncer and ROPecker.

For kBouncer, we show how all four of these attacks can be modified so kBouncer will not detect them.

We finally modify the one real-world exploit which ROPecker is shown to prevent to bypass ROPecker.

8.3.1 kBouncer Exploits

We modified four real-world exploits to bypass kBouncer. None of the modifications to these exploits took us significant effort. Once we were able to reproduce the exploit on our machine, each exploit took under half of a day's worth of work to make it bypass kBouncer. Given the long and difficult exploitation development process, we do not think this is meaningfully harder, especially for well-trained exploit developers.

MPlayer Lite r33063. This program [19] had a stack-based buffer overflow vulnerability, which was exploited by overwriting the SEH pointer [20]. The `avcodec-52.dll` does not have ASLR enabled. This `dll` is 10MB, and contains plenty of gadgets: there were 748 potential termination gadgets with two or fewer conditional branches. The first of these that we tried worked, and was given previously in Fig. 3(b).

[5] The most important deficiencies in our tool are as follows: we implement only the thirty most-used instructions (covering 99% of instructions used in our binaries), we ignore segment registers, we do not track several of the flags set by instructions, and we do not properly handle referencing variable register widths. Despite this, we have found our tool to be accurate in the vast majority of cases.

Adobe Reader 9.3.4. This Adobe Reader exploit uses a sophisticated JavaScript vulnerability and was built on the Metasploit framework [1]. This exploit relied on `icucnv36.dll` having ASLR disabled. This `dll` is 10MB and has 130 available termination gadgets with two or fewer conditional branches. We created a ROP chain to call `VirtualProtect` on a page and verified that code on this page in memory could be executed.

Adobe Flash 11.3.300. An integer overflow caused this vulnerability in Adobe Flash. This exploit was also built with the Metasploit framework [2]. The exploit relied on `msvcr71.dll` having ASLR disabled. This `dll` is 300KB and has 64 available termination gadgets. In this exploit, we were able to successfully change a page to be executable and spawn another process.

Internet Explorer 8. The final exploit we modified was in IE8 and also used Metasploit [3]. This exploit was the most difficult for us to modify, and required a manual stack-pivot to a controlled location so that we could invoke `VirtualProtect` in a call-preceded manner. We relied again on `msvcr71.dll` to spawn another process.

8.3.2 ROPecker Exploits

ROPecker was built as a Linux kernel module and was shown to stop two exploits. One of these two exploits by the authors is to exploit a 20-line example C program with a trivial stack overflow from ROPEME [17]. The other exploit is a real-world exploit in *hteditor*, which has a published vulnerability [33] they verified they defend against. Because they only evaluate their defense on one binary, we have only this one binary to demonstrate our attack on. We evaluate our two methods of attack (repeated history hiding and evasion attacks) on this binary.

The public vulnerability disclosure included an exploitable version of the *hteditor* source. We downloaded this and compiled it for our system with stack protection disabled, as we want to test how well ROPecker defends against attack, not how well stack canaries work.

Evasion attack. We successfully mounted an evasion attack on hteditor. Our exploit required 12 gadgets. We split the attack into two 10-gadget segments, with the second segment calling `execv` by overwriting the GOT entry for `strlen` and finding a call-preceded intended call to it.

Repeated history hiding attack. We successfully mounted a repeated history hiding attack on hteditor assuming four pages in the executable set. Our attack consisted of three phases. In the first two phases we computed the address of `execv`, and in the third we called it. In the first phase, we were able to use a gadget twice that we loaded once.

9 Related Work

Randomization-based approaches. Address Space Layout Randomization (ASLR) and Address Obfuscation [5] were first introduced to make it more difficult to inject shellcode, and were later applied to the text segment to prevent ROP attacks. Shacham et al. demonstrated a de-randomization attack [28] on PaX ASLR.

Address Space Layout Permutation (ASLP) [16] is similar in many ways to ASLR but provides higher entropy by permuting the locations of functions. Other defenses extends this further by randomizing the addresses of individual instructions [15, 31]. Another technique replaces short sequences of instructions with alternate, functionally-identical, equal-length sequence, hindering an attacker's ability to use unintended gadgets [22]. A recent just-in-time code reuse attack [29] compiles ROP on the fly to bypass ASLR.

Control-Flow Integrity (CFI). Abadi et al. introduced control-flow integrity (CFI) [4] as a method of preventing attacks by restricting jump, call, and return instructions to follow the statically-determined control-flow graph of the program. Due to the difficulty of obtaining a precise control-flow graph of the program, many defenses choose instead to enforce a less precise policy. Often, this policy simply requires that returns be call-preceded, and indirect calls point to the beginning of functions [34, 6, 32].

The attacks presented in this paper show these CFI based defenses are weaker than previously thought. Since call-preceded ROP is possible, most of these defenses can be broken with that technique alone. Concurrent to this work, a detailed examination of attacks on many CFI-based schemes came to this same conclusion [14].

Runtime defenses. There are many other types of defenses that can best be described as runtime defenses. DROP [9] monitors the runtime behavior of the process and, nearly identically to ROPecker, if there is a long consecutive sequence of returns, each of which contain fewer than a fixed length, the program is killed. Our work in this paper constitutes a total break of DROP. ROPGuard [13] contains several heuristics to detect ROP attacks. One of these is the call-preceded defense introduced earlier. ROPdefender [12] implements a shadow-stack and verifies that all returns exist somewhere on the shadow-stack. Our work does not apply to shadow-stack defenses.

Recompilation-based defenses. Other defenses rely on recompilation to remove gadget from the compiled binary. G-Free [21] does this by removing unintended return instructions and encrypting return addresses, so that `ret`-gadgets become nearly impossible to use. The return-less kernel [18] entirely removes the `c3` byte (the opcode of `ret`) from all instructions, and replaces valid returns with a lookup into a table containing the valid return sites.

10 Conclusion

In this paper, we have presented three building blocks for ROP attacks that allow us to break two state-of-the-art ROP defenses. We demonstrate the practicality of our attacks by modifying real-world exploits to bypass these defenses.

More broadly, our work disproves two pieces of conventional wisdom: that ROP attacks only consist of short gadgets, and that ROP attacks cannot be effectively mounted in call-preceded manner.

Future defenses must take care to guard against attacks similar to ours. Specifically, we suggest two particular requirements for future defenses. First, defenses should argue either that they can inspect all relevant past history or, if they have a limited history, that their limited view of history cannot be effectively cleared out by an attacker. Second, defenses that defend against one specific aspect of ROP must argue that is a *necessary* component of one.

We believe an important open research question is to determine what properties are truly *fundamental* about ROP attacks that are different than typical program execution. We hope future work will explore how these fundamental differences can be exploited to create general-purpose defenses.

Acknowledgments

We gratefully acknowledge Matthias Payer, Michael McCoyd, Thurston Dang, and the anonymous reviewers for their helpful feedback. This research was supported by Intel through the ISTC for Secure Computing, by the AFOSR under MURI award FA9550-12-1-0040 and MURI award FA9550-09-1-0539, and by the National Science Foundation under grant CCF-0424422.

References

[1] Adobe CoolType SING Table "uniqueName" Stack Buffer Overflow. `http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_cooltype_sing`.

[2] Adobe Flash Player 11.3 Kern Table Parsing Integer Overflow. `http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_flash_otf_font`.

[3] Microsoft Internet Explorer CButton Object Use-After-Free Vulnerability. `https://www.rapid7.com/db/modules/exploit/windows/browser/ie_cbutton_uaf`.

[4] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[5] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium*, volume 120, 2003.

[6] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[9] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.

[10] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, pages 12–12, 2005.

[11] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against rop attacks. NDSS14, 2014.

[12] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[13] Ivan Fratric and Elias Bachaalany. ROPGuard. `http://code.google.com/p/ropguard/`.

[14] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[15] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 571–585. IEEE, 2012.

[16] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[17] Long Le. Payload already inside: Data re-use for ROP exploits. *Black Hat USA*, 2010.

[18] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

[19] Nate M. MPlayer (r33064 Lite) Buffer Overflow + ROP exploit. `http://www.exploit-db.com/exploits/17124/`.

[20] Brian Mariani. Structured exception handler exploitation. `http://www.exploit-db.com/wp-content/themes/exploit/docs/17505.pdf`.

[21] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[22] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE, 2012.

[23] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[24] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *Security & Privacy, IEEE*, 2(4):20–27, 2004.

[25] Marco Prandini and Marco Ramilli. Return-oriented programming. *Security & Privacy, IEEE*, 10(6):84–87, 2012.

[26] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[27] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[28] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[29] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[30] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[31] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[32] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[33] ZadYree. HT Editor 2.0.20 Buffer Overflow (ROP PoC). `http://www.exploit-db.com/exploits/22683/`.

[34] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection

Lucas Davi, Ahmad-Reza Sadeghi
Intel CRI-SC, TU Darmstadt, Germany

Daniel Lehmann
TU Darmstadt, Germany

Fabian Monrose
University of North Carolina at Chapel Hill, USA

Abstract

Return-oriented programming (ROP) offers a robust attack technique that has, not surprisingly, been extensively used to exploit bugs in modern software programs (e.g., web browsers and PDF readers). ROP attacks require no code injection, and have already been shown to be powerful enough to bypass fine-grained memory randomization (ASLR) defenses. To counter this ingenious attack strategy, several proposals for enforcement of (coarse-grained) control-flow integrity (CFI) have emerged. The key argument put forth by these works is that coarse-grained CFI policies are sufficient to prevent ROP attacks. As this reasoning has gained traction, ideas put forth in these proposals have even been incorporated into coarse-grained CFI defenses in widely adopted tools (e.g., Microsoft's EMET framework).

In this paper, we provide the first comprehensive security analysis of various CFI solutions (covering kBouncer, ROPecker, CFI for COTS binaries, ROPGuard, and Microsoft EMET 4.1). A key contribution is in demonstrating that these techniques can be effectively undermined, even under weak adversarial assumptions. More specifically, we show that with bare minimum assumptions, turing-complete and real-world ROP attacks can still be launched even when the strictest of enforcement policies is in use. To do so, we introduce several new ROP attack primitives, and demonstrate the practicality of our approach by transforming existing real-world exploits into more stealthy attacks that bypass coarse-grained CFI defenses.

1 Introduction

Today, runtime attacks remain one of the most prevalent attack vectors against software programs. The continued success of these attacks can be attributed to the fact that large portions of software programs are implemented in type-unsafe languages (C, C++, or Objective-C) that do not enforce bounds checking on data inputs. Moreover, even type-safe languages (e.g., Java) rely on interpreters (e.g., the Java virtual machine) that are in turn implemented in type-unsafe languages.

Sadly, as modern compilers and applications become more and more complex, memory errors and vulnerabilities will likely continue to persist, with little end in sight [41]. The most prominent example of a memory error is the stack overflow vulnerability, where the adversary overflows a local buffer on the stack and overwrites a function's return address [4]. While today's defenses protect against this attack strategy (e.g., by using stack canaries [15]), other avenues for exploitation exists, including those that leverage heap [33], format string [21], or integer overflow [6] vulnerabilities.

Regardless of the attacker's method of choice, exploiting a vulnerability and gaining control over an application's control-flow is only the first step of a runtime attack. The second step is to launch malicious program actions. Traditionally, this has been realized by injecting malicious code into the application's address space, and later executing the injected code. However, with the wide-spread enforcement of the non-executable memory principle (called data execution prevention in Windows) such attacks are more difficult to do today [28]. Unfortunately, the long-held assumption that only new injected code bared risks was shattered with the introduction of code reuse attacks, such as return-into-libc [30, 37] and return-oriented programming (ROP) [35]. As the name implies, code reuse attacks do not require any code injection and instead use code already resident in memory.

One of the most promising defense mechanisms against such runtime attacks is the enforcement of *control-flow integrity (CFI)* [1, 3]. The main idea of CFI is to derive an application's control-flow graph (CFG) prior to execution, and then monitor its runtime behavior to ensure that the control-flow follows a legitimate path of the CFG. Any deviation from the CFG leads to a CFI exception and subsequent termination of the application.

Although CFI requires no source code of an application, it suffers from practical limitations that impede

its deployment in practice, including significant performance overhead of 21%, on average [3, Section 5.4], when function returns are validated based on a return address (shadow) stack. To date, several CFI frameworks have been proposed that tackle the practical shortcomings of the original CFI approach. ROPecker [13] and kBouncer [31], for example, leverage the branch history table of modern x86 processors to perform a CFI check on a short history of executed branches. More recently, Zhang and Sekar [46] demonstrate a new CFI binary instrumentation approach that can be applied to commercial off-the-shelf binaries.

However, the benefits of these state-of-the-art solutions comes at the price of relaxing the original CFI policy. Abstractly speaking, coarse-grained CFI allows for CFG relaxations that contains dozens of more legal execution paths than would be allowed under the approach first suggested by Abadi et al. [3]. The most notable difference is that the coarse-grained CFI policy for return instructions only validates if the return address points to an instruction that follows after a call instruction. In contrast, Abadi et al. [3]'s policy for fine-grained CFI ensures that the return address points to the original caller of a function (based on a shadow stack). That is, a function return is only allowed to return to its original caller.

Surprisingly, even given these relaxed assumptions, all recent coarse-grained CFI solutions we are aware of claim that their relaxed policies are sufficient to thwart ROP attacks[1]. In particular, they claim that the property of Turing-completeness is lost due to the fact that the code base which an adversary can exploit is significantly reduced. Yet, to date, no evidence substantiating these assertions has been given, raising questions with regards to the true effectiveness of these solutions.

Contribution. We revisit the assumption that coarse-grained CFI offers an effective defense against ROP. For this, we conduct a security analysis of the recently proposed CFI solutions including kBouncer [31], ROPecker [13], CFI for COTS binaries [46], ROPGuard [20], and Microsofts' EMET tool [29]. In particular, we derived a combined CFI policy that takes for each indirect branch class (i.e., return, indirect jump, indirect call) and behavioral-based heuristics (e.g., the number of instruction executed between two indirect branches), the most restrictive setting among these policies. Afterwards, we use our combined CFI policy and a weak adversary having access to only a *single* — and common used system library — to realize a Turing-complete gadget set. The reduced code base mandated that we develop several new return-oriented programming attack gadgets to facilitate our attacks. To demonstrate the power of our attacks, we show how to harden existing real-world exploits against the Windows version of Adobe Reader [26] and mPlayer [10] so that they bypass coarse-grain CFI protections. We also demonstrate a proof-of-concept attack against a Linux-based system.

2 Background

2.1 Return-Oriented Programming

Return-oriented programming (ROP) belongs to the class of runtime attacks that require no code injection. The basic idea is to combine short code sequences already residing in the address space of an application (e.g., shared libraries and the executable itself) to perform malicious actions. Like any other runtime attack, it first exploits a vulnerability in the software running on the targeted system. Relevant vulnerabilities are memory errors (e.g., stack, heap, or integer overflows [33]) which can be discovered by reverse-engineering the target program. Once a vulnerability has been discovered, the adversary needs to exploit it by providing a malicious input to the program, the so-called ROP payload. The applicability of ROP has been shown on many platforms including x86 [35], SPARC [7], and ARM [27].

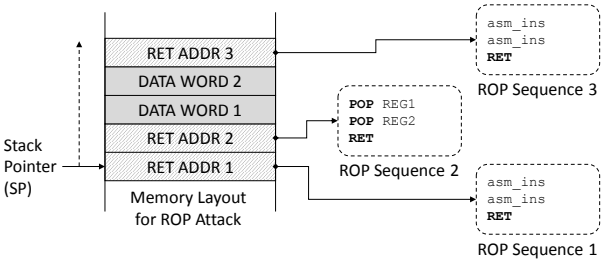


Figure 1: Memory snapshot of a ROP Attack

An example ROP payload and a typical memory layout for a ROP attack is shown in Figure 1. Basically, the ROP payload consists of a number of return addresses each pointing to a short code sequence. These sequences consist of a small number of assembler instructions (denoted in Figure 1 as `asm_ins`), and traditionally terminate in a return [35] instruction[2]. The indirect branches are responsible for chaining and executing one ROP sequence after the other.

In addition to return addresses, the adversary writes several data-words in memory that are used by the invoked code sequences (usually via stack POP instructions as shown in ROP Sequence 2). At the beginning of the attack, the stack pointer (SP) points to the first return address of the payload. Once the first sequence has been executed, its final return instruction (RET) advances the stack pointer by one memory word, loads the next return address from the stack, and transfers the control-flow to the next code sequence.

The combination of the invoked ROP sequences induce the malicious operations. Typically, these sequences are identified within an (offline) static analy-

sis phase on the target program binary and its linked shared libraries. Furthermore, one or multiple ROP sequences can form a *gadget*, where a gadget accomplishes a specific task such as adding two values or storing a data word into memory. These gadgets typically form a Turing-complete language meaning that an adversary can perform arbitrary (malicious) computation.

A well-known defense against ROP is address space layout randomization (ASLR) which randomizes the base address of libraries and executables, thereby randomizing the start addresses of code sequences needed by the adversary in her ROP attack. However, ASLR is vulnerable to memory disclosure attacks, which reveal runtime addresses to the adversary. Memory disclosure can even be exploited to circumvent fine-grained ASLR schemes, where the location of each code block is randomized in memory by identifying ROP gadgets on-the-fly and generating a ROP payload at runtime [36].

2.2 Control-Flow Integrity

Although W⊕X, ASLR and other protection mechanisms have been widely adopted, their security benefits remain open to debate [1]. The main critique is the lack of a clear attack model and formal reasoning. To address this, Abadi et al. [3] proposed a new security property called control-flow integrity (CFI). A program maintains CFI if its path of execution adheres to a certain pre-defined control-flow graph (CFG). This CFG consists of basic blocks (BBLs) as nodes, where a BBL is a sequence of assembler instructions. Edges connect two nodes, whenever the program may legally transfer control-flow from one to the next BBL. A control-flow transfer may be either a direct or indirect branch instruction (e.g., call, jump, or return). To ensure that a program follows a valid path in the CFG, CFI inserts labels at the beginning of basic blocks. Whenever there is a control-flow transfer at runtime, CFI validates whether the indirect branch targets a BBL with a valid label.

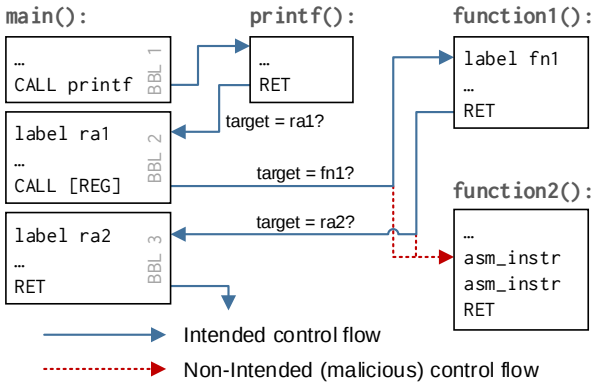


Figure 2: The CFG shepherds control-flow transfers

An example for CFI enforcement is shown in Figure 2. It shows a program consisting of a main function that invokes directly the library function *printf()*, and indirectly the local subroutine *function1()*. The indirect call to *function1()* in BBL 2 is critical, since an adversary may load an arbitrary address into the register by means of a buffer overflow attack. However, the CFG states that this indirect call is only allowed to target *function1()*. Hence, at runtime, CFI validates whether the indirect call in BBL 2 is targeting label **fn1**. If an adversary aims to redirect the call to a code sequence residing in *function2()*, CFI will prevent this malicious control-flow, because label **fn1** is not defined for *function2()*. Similarly, CFI protects the return instructions of *printf()* and *function1()*, which an adversary could both exploit by overwriting a return address on the stack. The specific CFI checks in Figure 2 validate if the returns address label **ra1** or **ra2**, respectively.

It is also prudent to note that CFI has been studied in many domains. For instance, it has been used as an enabling technology for software fault isolation by Abadi et al. [2] and Yee et al. [43]. CFI enforcement has also been shown for hypervisors [42], commodity operating system kernels [16] and mobile devices [18]. In other communities, Zeng et al. [44] and Pewny and Holz [32], for example, have shown how to instrument a compiler to generate CFI-protected applications. Lastly, Budiu et al. [8] have explored architectural support to tackle the performance overheads of software-only based solutions.

2.3 Control-Flow Integrity Challenges

There are several factors that impede the deployment of control-flow integrity (CFI) in practice, including those related to control-flow graph (CFG) coverage, performance, robustness, and ease of deployment.

Before proceeding further, we note that besides presenting the design of CFI, Abadi et al. [3] also included a formal security proof for the soundness of their solution. A key observation noted in that work is that "despite attack steps, the program counter always follows the CFG." [3, p. 4:34]. In other words, in Abadi et al. [3], every control-flow is permitted as long as the CFG allows it. Consequently, the quality of protection from control-flow attacks rests squarely on the level of CFG coverage. And that is exactly where recent CFI solutions have deviated (substantially) from the original work, primarily as a means to address performance issues.

Recall that in the original proposal, the CFG was obtained a priori using binary analysis techniques supported by a proprietary framework called Vulcan. Since the CFG is created ahead of time, it is not capable of capturing the dynamic nature of the call stack. That is, with only the CFG at hand, one can not enforce that functions return to their most recent call site, but only that they re-

turn to any of the possible call sites. This limitation is tackled by adding a shadow stack to the statically created CFG. Intuitively, upon each call, the return address is placed in a safe location in memory, so that an instrumented return is able to compare the return address on the stack with one on a shadow stack, and the program is terminated if a deviation is detected [3, 14, 18]. In this way, many control-flow transfers are prohibited, largely reducing the gadget space available for a return-oriented programming attack.

Given the power of CFI, it is surprising that it has not yet received widespread adoption. The reason lies in the fact that extracting the CFG is not as simple as it may appear. To see why, notice that (1) source code is not readily available (thereby limiting compiler-based approaches), (2) binaries typically lack the necessary debug or relocation information, as was needed for example, in the Vulcan framework, and (3) the approach induces high performance overhead due to dynamic rewriting and runtime checks. Much of the academic research on CFI in the last few years has focused on techniques for tackling these drawbacks.

3 Categorizing Coarse-Grained Control-Flow Integrity Approaches

As noted above, a number of new control-flow integrity (CFI) solutions have been recently proposed to address the challenges of good runtime performance, high robustness and ease of deployment. The most prominent examples include kBouncer [31], ROPecker [13], CFI for COTS binaries [46], and ROPGuard [20]. To aide in better understanding the strenghts and limitations of these proposals, we first provide a taxonomy of the various CFI policies embodied in these works. Later, to strengthen our own analyses, we also derive a combined CFI policy that takes into account the most restrictive CFI policy.

3.1 CFI Policies

Table 1 summarizes the five CFI policies we use throughout this paper to analyze the effectiveness of coarse-grained CFI solutions. Specifically, we distinguish between three types of policies, namely ① policies used for indirect branch instructions, ② general CFI heuristics that do not provide well-founded control-flow checks but instead try to capture general machine state patterns of ROP attacks and ③ a policy class that covers the time CFI checks are enforced.

We believe this categorization covers the most important aspects of CFI-based defenses suggested to date. In particular, they cover polices for each indirect branch the processor supports since all control-flow attacks (including ROP) require exploiting indirect branches. Second, heuristics are used by several coarse-grained CFI approaches (e.g., [20, 31]) to allow more relaxed CFI policies for indirect branches. Finally, the time-of-check policy is an important aspect, because it states at which execution state ROP attacks can be detected. We elaborate further on each of these categories below.

Category	Policy	x86 Example	Description
①	CFI_{RET}	`ret`	returns
	CFI_{JMP}	`jmp reg|mem`	indirect jumps
	CFI_{CALL}	`call reg|mem`	indirect calls
②	CFI_{HEU}		heuristics
③	CFI_{TOC}		time of CFI check

Table 1: Our CFI policies

1 – Indirect Branches. Recall that the goal of CFI is to validate the control-flow path taken at *indirect* branches, i.e., at those control-flow instructions that take the target address from either a processor register or from a data memory area[3]. The indirect branch instructions present on an Intel x86 platform are indirect calls, indirect jumps, and returns. Since CFI solutions apply different policies for each type of indirect branch, it is only natural that there are three CFI policies in this category, denoted as CFI_{CALL} (indirect function calls), CFI_{JMP} (indirect jumps), CFI_{RET} (function returns).

2 – Behavior-Based Heuristics (HEU). Apart from enforcing specific policies on indirect branch instructions, CFI solutions can also validate other program behavior to detect ROP attacks. One prominent example is the number of instructions executed between two consecutive indirect branches. The expectation is that the number of such instructions will be low (compared to ordinary execution) because ROP attacks invoke a chain of short code sequences each terminating in an indirect branch instruction.

3 – Time of CFI Check (TOC). Abadi et al. argued that a CFI validation routine should be invoked whenever the program issues an indirect branch instruction [3]. In practice, however, doing so induces significant performance overhead. For that reason, some of the more recent CFI approaches reduce the number of runtime checks, and only enforce CFI validation at critical program states, e.g., before a system or API call.

3.2 Instantiation in Recent Proposals

Next, we turn our attention to the specifics of how these policies are implemented in recent CFI mechanisms.

3.2.1 kBouncer

The approach of Pappas et al. [31], called kBouncer, deploys techniques that fall in each of the aforementioned categories. Under category ①, Pappas et al. [31] leverage the x86-model register set called last branch record (LBR). The LBR provides a register set that holds the

last 16 branches the processor has executed. Each branch is stored as a pair consisting of its source and target address. kBouncer performs CFI validation on the LBR entries whenever a Windows API call is invoked. Its promise resides in the fact that these checks induce almost no performance overhead, and can be directly applied to existing software programs.

With respect to its policy for returns, kBouncer identifies those LBR entries whose source address belong to a return instruction. For these entries, kBouncer checks whether the target address (i.e., the return address) points to a *call-preceded* instruction. A call-preceded instruction is any instruction in the address space of the application that follows a call instruction. Internally, kBouncer disassembles a few bytes before the target address and terminates the process if it fails to find a call instruction.

While kBouncer does not enforce any CFI check on indirect calls and jumps, Pappas et al. [31] propose behavioral-based heuristics (category ②) to mitigate ROP attacks. In particular, the number of instructions executed between consecutive indirect branches (i.e., "the sequence length") is checked, and a limit is placed on the number of sequences that can be executed in a row.[4]

A key observation by Pappas et al. [31] is that even though pure ROP payloads can perform Turing-complete computation, in actual exploits they will ultimately need to interact with the operating system to perform a meaningful task. Hence, as a time-of-CFI check policy (category ③) kBouncer instruments and places hooks at the entry of a WinAPI function. Additionally, it writes a *checkpoint* after CFI validation to prohibit an adversary from simply jumping over the hook in userspace.

3.2.2 ROPGuard and Microsoft EMET

Similar to Pappas et al. [31], the approach suggested by Fratric [20] (called ROPGuard) performs CFI validation when a critical Windows function is called. However, its policies differ from that of Pappas et al. [31].

First, with respect to policies under category ①, upon entering a critical function, ROPGuard validates whether the return address of that critical function points to a call-preceded instruction. Hence, it prevents an adversary from using a ROP sequence terminating in a return instruction to invoke the critical Windows function. In addition, ROPGuard checks if the memory word before the return address is the start address of the critical function. This would indicate that the function has been entered via a return instruction. ROPGuard also inspects the stack and predicts future execution to identify ROP gadgets. Specifically, it walks the stack to find return addresses. If any of these return addresses points to a non call-preceded instruction, the program is terminated.

Interestingly, there is no CFI policy for indirect calls or indirect jumps. Furthermore, ROPGuard's only heuristic under category ② is for validating that the stack pointer does not point to a memory location beyond the stack boundaries. While doing so prevents ROP payload execution on the heap, it does not prevent traditional stack-based ROP attacks; thus the adversary could easily reset the stack pointer before a critical function is called.

Remarks: ROPGuard and its implementation in Microsoft EMET [5] use similar CFI policies as in kBouncer. One difference is that kBouncer checks the indirect branches executed in the past, while ROPGuard only checks the current return address of the critical function, and for future execution of ROP gadgets. ROPGuard is vulnerable to ROP attacks that are capable of jumping over the CFI policy hooks, and cannot prevent ROP attacks that do not attempt to call any critical Windows function. To tackle the former problem (i.e., bypassing the policy hook), EMET adds some randomness in the length and structure of the policy hook instructions. Hence, the adversary has to guess the right offset to successfully deploy her attack. However, recent memory disclosure attacks show that such randomization approaches can be easily circumvented [36].

3.2.3 ROPecker

ROPecker is a linux-based approach suggested by Cheng et al. [13] that also leverages the last branch record register set to detect past execution of ROP gadgets. Moreover, it speculatively emulates the future program execution to detect ROP gadgets that will be invoked in the near future. To accomplish this, a static offline phase is required to generate a database of all possible ROP code sequences. To limit false positives, Cheng et al. [13] suggest that only code sequences that terminate after at most n instructions in an indirect branch should be recorded.

For its policies in category ①, ROPecker inspects each LBR entry to identify indirect branches that have redirected the control-flow to a ROP gadget. This decision is based on the gadget database that ROPecker derived in the static analysis phase. ROPecker also inspects the program stack to predict future execution of ROP gadgets. There is no direct policy check for indirect branches, but instead, possible gadgets are detected via a heuristic. More specifically, the robustness of its behavioral-based heuristic (category ②) completely hinges on the assumption that ROP code sequences will be short and that there will always be a chain of at least some threshold number of consecutive ROP sequences.

Lastly, its time of CFI check policy (category ③) is triggered whenever the program execution leaves a sliding window of two memory pages.

Remarks: Clearly, ROPecker performs more frequently CFI checks than both kBouncer and ROPGuard. Hence, it can detect ROP attacks that do not necessar-

ily invoke critical functions. However, as we shall show later, the fact that there is no policy for the target of indirect branches is a significant limitation.

3.2.4 CFI for COTS Binaries

Most closely related to the original CFI work by Abadi et al. [3] is the proposal of Zhang and Sekar [46] which suggest an approach for commercial-off-the-shelf (COTS) binaries based on a static binary rewriting approach, but without requiring debug symbols or relocation information of the target application. In contrast to all the other approaches we are aware of, the CFI checks are directly incorporated into the application binary. To do so, the binary is disassembled using the Linux disassembler objdump. However, since that disassembler uses a simple linear sweep disassembly algorithm, Zhang and Sekar [46] suggest several error correction methods to ensure correct disassembly. Moreover, potential candidates of indirect control-flow target addresses are collected and recorded. These addresses comprise possible return addresses (i.e., call-preceded instructions), constant code pointers (including memory locations of pointers to external library calls), and computed code pointers (used for instance in switch-case statements). Afterwards, all indirect branch instructions are instrumented by means of a jump to a CFI validation routine.

Like the aforementioned works, the approach of Zhang and Sekar [46] checks whether a return or an indirect jump targets a call-preceded instruction. Furthermore, it also allows returns and indirect jumps to target any of the constant and computed code pointers, as well as exception handling addresses. Hence, the CFI policy for returns is not as strict as in kBouncer, where only call-preceded instructions are allowed. On the other hand, their approach deploys a CFI policy for indirect jumps, which is largely unmonitored in the other approaches. However, it does not deploy any behavioral-based heuristics (category ②).

Lastly, CFI validation (category ③) is performed whenever an indirect branch instruction is executed. Hence, it has the highest frequency of CFI validation invocation among all discussed CFI approaches.

Similar CFI policies are also enforced by CCFIR (compact CFI and randomization) [45]. In contrast to CFI for COTS binaries, all control-flow targets for indirect branches are collected and randomly allocated on a so-called springboard section. Indirect branches are only allowed to use control-flow targets contained in that springboard section. Specifically, CCFIR enforces that returns target a call-preceded instruction, and indirect calls and jumps target a previously collected function pointer. Although the randomization of control-flow targets in the springboard section adds an additional layer of security, it is not directly relevant for our analysis, since memory disclosure attacks can reveal the content of the entire springboard section [36]. The CFI policies enforced by CCFIR are in principle covered by CFI for COTS binaries. However, there is one noteworthy policy addition: CCFIR denies indirect calls and jumps to target pre-defined sensitive functions (e.g., *VirtualProtect*). We do not consider this policy for two reasons: first, this policy violates the default external library call dispatching mechanism in Linux systems. Any application linking to such a sensitive (external) function will use an indirect jump to invoke it.[5] Second, as shown in detail by Göktas et al. [22] there are sufficient direct calls to sensitive functions in Windows libraries which an adversary can exploit to legitimately transfer control to a sensitive function.

Remarks: The approach of Zhang and Sekar [46] is most similar to Abadi et al. [3]'s original proposal in that it enforces CFI policies each time an indirect branch is invoked. However, to achieve better performance and to support COTS binaries, it deploys less fine-grained CFI policies. Alas, its coarse-grain policies allow one to bypass the restrictions for indirect call instructions (CFI_{CALL}). The main problem is caused by the fact that the integrity of indirect call pointers is not validated. Instead, it is only enforced that an indirect call takes a pointer from a memory location that is expected to hold indirect call targets. A typical example is the Linux global offset table (GOT) which holds the target addresses for library calls. This leaves the solution vulnerable to so-called GOT-overwrite attacks [9] that overwrite pointers (in the GOT) to external library calls. We return to this vulnerability in §5. Moreover, even if one would ensure the integrity of these pointers, we are still allowed to use a valid code pointer defined in the external symbols. Hence, the adversary can invoke dangerous functions such as *VirtualAlloc()* and *memcpy()* that are frequently used in applications and libraries.

3.3 Deriving a Combined CFI Policy

In our analysis that follows, we endeavor to have the best possible protections offered by the aforementioned CFI mechanisms in place at the time of our evaluation. Therefore, our combined CFI policy (see Table 2) selects the most restrictive setting for each policy. Nevertheless, despite this combined CFI policy, we then show that one can still circumvent these coarse-grained CFI solutions, construct Turing-complete ROP attacks (under realistic assumptions) and launch real-world exploits.

At this point, we believe it is prudent to comment on the parameter choices in these prior works — and that adopted in Table 2. In particular, one might argue that the prerequisite thresholds could be adjusted to make ROP attacks more difficult. To that end, we note that Pappas et al. [31] performed an extensive analysis to arrive at the

Control-Flow Integrity (CFI) Policies	CFI for COTS [46]	kBouncer [31]	ROPecker [13]	ROPGuard [20]	EMET 4.1 [29]	*Combined Policy*
CFI_{RET} : destination has to be call-preceded	✓	✓	◯	✓	✓	✓
CFI_{RET} : destination can be taken from a code pointer	✓	✗	◯	✗	✗	✗
CFI_{JMP} : destination has to be call-preceded	✓	◯	◯	◯	◯	✓
CFI_{JMP} : destination can be taken from a code pointer	✓	◯	◯	◯	◯	✓
CFI_{CALL} : destination can be taken from an exported symbol	✓	◯	◯	◯	◯	✓
CFI_{CALL} : destination can be taken from a code pointer	✓	◯	◯	◯	◯	✓
CFI_{HEU} : allow only s consecutive short sequences,	◯	$s <= 7$	$s <= 10$	◯	◯	$s <= 7$
CFI_{HEU} : where *short* is defined as n instructions	◯	$n <= 20$	$n <= 6$	◯	◯	$n <= 20$
CFI_{TOC} : check at every indirect branch	✓	◯	◯	◯	◯	Always observed
CFI_{TOC} : check at critical API functions or system calls	◯	✓	✓	✓	✓	
CFI_{TOC} : check when leaving sliding code window	◯	◯	✓	◯	◯	

Table 2: Policy comparison of coarse-grained CFI solutions: ✓ indicates that the CFI policy is applied and enforced. ✗ means that the CFI policy is prohibited (corresponding execution flows would lead to an attack alarm) . ◯ indicates that the CFI policy is not applied/enforced. The combined policy takes the most restrictive setting for each CFI policy.

best range of thresholds for the recommended number of consecutive short sequences (s) with a given sequence length of $n <= 20$. Their analysis reveals that adjusting the thresholds for s beyond their recommended values is hardly realistic: when every function call was instrumented, 975 false positives were recorded for $s <= 8$.

An alternative is to increase the sequence length n (e.g., setting it to $n <= 40$). Doing so would require an adversary to find a long sequence of 40 instructions after each seventh short sequence (for $s <= 7$). However, increasing the threshold for the sequence length will only exacerbate the false positive issue. For this reason, Pappas et al. [31] did not consider sequences consisting of more than 20 instructions as a gadget in their analyses. We provide our own assessment in §5.3.

The approach of Cheng et al. [13], on the other hand, uses different thresholds for s and n than in kBouncer. Making the thresholds in ROPecker more conservative (e.g., reducing s and increasing n) will lead to the same false positives problems as in kBouncer. Moreover, the problem would be worse, since ROPecker performs CFI validation more frequently than kBouncer. Nevertheless, we show that regardless of the specific choice of parameter chosen in the recommended ranges, our attacks render these defenses ineffective in practice (see Section 5).

4 Turing-Complete ROP Gadget Set

We now explore whether or not it is possible to derive a Turing-complete gadget set even when all state-of-the-art coarse-grained CFI protections are enforced. In particular, we desire a gadget set that still allows an adversary to undermine the combined CFI policy (see Table 2).

Assumptions. To be as pragmatic as possible, we assume that the adversary can only leverage the presence of a single shared library to derive the gadget set. This is a very stringent requirement placed on ourselves since modern programs typically link to dozens of libraries.

Note also that we are not concerned with circumventing other runtime protection mechanisms such as ASLR or stack canaries. The reasons are twofold: first, coarse-grained CFI protection approaches do not rely on the presence of other defenses to mitigate against code reuse attacks. Second, in contrast to CFI, ASLR and protection mechanisms that defend against code pointer overwrites (e.g., stack canaries, bounds checkers, pointer encryption) do not offer a general defense, and moreover, are typically bypassed in practice. In particular, ASLR is vulnerable to memory disclosure attacks [36, 38]. That said, the attacks and return-oriented programming gadgets we present in the following can be also leveraged to mount memory disclosure attacks in the first stage.

Methodology and Outline. Our analysis is performed primarily on Windows as it is the most widely deployed desktop operating system today. Specifically, we inspect `kernel32.dll` (on x86 Windows 7 SP1), a 848kb system library that exposes Windows API functions and is by default linked to nearly every major Windows process (e.g., Adober Reader, IE, Firefox, MS Office). It

is also noteworthy to mention that our results do not only apply to Windows; Although we did not perform a Turing-complete gadget analysis for Linux's default library (`libc.so`), to demonstrate the generality of our approach, we provide a shellcode exploit that uses gadgets from `libc` (see §5.2). To facilitate the gadget finding process, we developed a static analysis python module in `IDA Pro` that outputs all call-preceded sequences ending in an indirect branch. We also developed a sequence filter in the general purpose `D` programming language that allows us to check for sequences containing a specific register, instruction, or memory operand. Note that in the subsequent discussions, we use the Intel assembler syntax, e.g., `mov destination, source`, and use a semicolon to separate two consecutive instructions.

We first review in §4.1 the basic gadgets that form a Turing-complete language [12, 35]. To achieve Turing-completeness, we require gadgets to realize memory load and store operations, as well as a gadget to realize a conditional branch. Afterwards, we present two new gadget types called the Call-Ret-Pair gadget (§4.2.1) and the Long-NOP gadget (§4.2.2). Constructing the latter was a non-trivial engineering task and the outcome played an important role in "stitching" gadgets together, thereby bypassing coarse-grained CFI defenses. It should also be noted that we only present a *subset* of the available sequences. Eliminating the specific few sequences presented here will not prevent our attack, since `kernel32.dll` (and many other libraries) provides a multitude of other sequences we could have leveraged.

4.1 Basic Gadget Arsenal

Loading Registers. Load gadgets are leveraged in nearly every ROP exploit to load a value from the stack into a CPU register. Recall that x86 provides six general registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`), a base/frame pointer register (`ebp`), the stack pointer (`esp`), and the instruction pointer (`eip`). All registers can be directly accessed (read and write) by assembler instructions except the `eip` which is only indirectly influenced by dedicated branch instructions such as `ret`, `call`, and `jmp`.

Typically, stack loading is achieved on x86 via the `POP` instruction. The call-preceded load gadgets we identified in `kernel32.dll` are summarized in Table 3. Except for the `ebp` register, we are not able to load any other register without inducing a side-effect, i.e., without affecting other registers. That said, notice that the sequence for `esi`, `edi`, and `ecx` only modifies the base pointer (`ebp`). Because traditionally `ebp` holds the base pointer and no data, and ordinary programs can be compiled without using a base pointer, we consider `ebp` as an *intermediate register* in our gadget set. The astute reader would have noticed that the sequences for `edi` and `ecx` modify the stack pointer as well through the `leave` instruction, where `leave` behaves as `mov esp,ebp; pop ebp`. However, we can handle this side-effect, since the stack pointer receives the value from our intermediate register `ebp`. Hence, we first invoke the load gadget for `ebp` and load the desired stack pointer value, and afterwards call the sequence for `edi`/`ecx`.

More challenges arise when loading the general-purpose registers `eax`, `ebx`, and `edx`. While `ebx` can be loaded with side-effects, we were not able to find any useful stack pop sequence for `eax` and `edx`. This is not surprising given the fact that we must use call-preceded sequences. Typically, these sequences are found in function epilogues, where a function epilogue is responsible for resetting the caller-saved registers (`esi`, `edi`, `epb`). We alleviate the side-effects for `ebx` by loading all the caller-saved registers from the stack.

Register	Call-Preceded Sequence (ending in `ret`)
EBP	`pop ebp`
ESI	`pop esi; pop ebp`
EDI	`pop edi; leave`
ECX	`pop ecx; leave`
EBX	`pop edi; pop esi; pop ebx; pop ebp`
EAX	`mov eax,edi; pop edi; leave`
EDX	`mov eax,[ebp-8]; mov edx,[ebp-4]; pop edi; leave`

Table 3: Register Load Gadgets

For `eax` and `edx`, data movement gadgets can be used. As can be seen in Table 3, `eax` can be loaded using the `edi` load gadget in advance. The situation is more complicated for `edx`, especially given our choice to only use `kernel32.dll`. In particular, while there is a sequence that allows one to load `edx` by using the `ebp` load gadget beforehand, it is challenging to do so since the adversary would need to save the state of some registers. That said, other default Windows libraries (such as `shell32.dll`) offer several more convenient gadgets to load `edx` (e.g., a common sequence we observed was `pop edx; pop ecx; jmp eax`), and so this limitation should not be a major obstacle in practice.

Loading and Storing from Memory. In general, software programs can only accomplish their tasks if the underlying processor architecture provides instructions for loading from memory and storing values to memory. Similarly, ROP attacks require memory load and store gadgets. Although we have found several load and store gadgets, we focus on the gadgets listed in Table 4.

In particular, we discovered load gadgets that use `eax` as the destination register. The specific load gadget shown in Table 4 loads a value from memory pointed to by `ebp+8`. Hence, the adversary is required to correctly set the target address of the memory load operation in `ebp` via the register load gadget shown in Table 3.

Type	Call-Preceded Sequence (ending in `ret`)
LOAD (eax)	`mov eax, [ebp+8]; pop ebp`
STORE (eax)	`mov [esi],eax; xor eax,eax;` `pop esi; pop ebp`
STORE (esi)	`mov [ebp-20h],esi`
STORE (edi)	`mov [ebp-20h],edi`

Table 4: Selected Memory Load and Store Gadgets

We also identified a corresponding memory store gadget on `eax`. The shown gadget stores `eax` at the address provided by register `esi`, which needs to be initialized by a load register gadget beforehand. The gadget has no side-effects, since it resets `eax` (which was stored earlier) and loads new values from the stack into `esi` (which held the target address) and `ebp` (our intermediate register).

Given a memory store gadget for `eax` and the fact that we have already identified register load gadgets for each register, it is sufficient to use the same memory load on `eax` to load any other register. This is possible because we use the `eax` load gadget to load the desired value from memory, store it afterwards on the stack, and finally use one of the register load gadgets to load the value into the desired register. Finally, we also identified some convenient memory store gadgets for `esi` and `edi` only requiring `ebp` to hold the target address of the store operation.

Arithmetic and Logical Gadgets. For arithmetic operations we utilize the sequence containing the x86 `sub` instruction shown in Table 5. This instruction takes the operands from `eax` and `esi` and stores the result of the subtraction into `eax`. Both operands can be loaded by using the register load gadgets (see Table 3). The same gadget can be used to perform an addition: one only needs to load the two's complement into `esi`. Based on addition and subtraction, we can realize multiplication and division as well. Unfortunately, logical gadgets are not as commonplace. There is, however, a XOR gadget that takes its operands from `eax` and `edi` (see Table 3).

Type	Call-Preceded Sequence (ending in `ret`)
ADD/SUB	`sub eax,esi; pop esi; pop ebp`
XOR	`xor eax,edi; pop edi; pop esi;` `pop ebp`

Table 5: Arithmetic and Logical Gadgets

Branching Gadgets. We remind the reader that branching in ROP attacks is realized by modifying the stack pointer rather than the instruction pointer [35]. In general, we can distinguish two different types of branches: unconditional and conditional branches. `kernel32.dll`, for example, offers two variants for a unconditional branch gadget (see Table 6). The first uses the `leave` instruction to load the stack pointer (`esp`) with a new address that has been loaded before into our intermediate register `ebp`. The second variant realizes the unconditional branch by adding a constant offset to `esp`. Either one suffices for our purposes.

Type	Call-Preceded Sequence (ending in `ret`)
unconditional branch 1	`leave`
unconditional branch 2	`add esp,0Ch; pop ebp`
conditional LOAD(eax)	`neg eax; sbb eax,eax;` `and eax,[ebp-4];leave`

Table 6: Branching Gadgets

Conditional branch gadgets change the stack pointer iff a particular condition holds. Because load, store, and arithmetic/logic computation can be conveniently done for `eax`, we could place the conditional in this register. Unfortunately, because a direct load of `esp` (that depended on the value of `eax`) was not readily available, we realized the conditional branch in three steps requiring the invocation of only four ROP sequences. That said, our gadget is still within the constraints for the number of allowable consecutive sequences in the Combined CFI-enforcement Policy (see $n = 8$ for CFI_{HEU} in Table 2).

First, we use the conditional branch gadget (see Table 6) to either load `0` or a prepared value into `eax`. In this sequence `neg eax` computes the two's complement and, more importantly, sets the carry flag to zero if and only if `eax` was zero beforehand. This is nicely used by the subsequent `sbb` instruction, which subtracts the register from itself, always yielding zero, but additionally subtracting an extra one if the carry flag is set. Because subtracting one from zero gives `0xFFFFFFFF`, the next `and` masks either none or all the bits. Hence, the result in `eax` will be exactly the contents of `[ebp-4]` if `eax` was zero, or zero otherwise. One might think that it is very unlikely to find sequences that follow the pattern `neg-sbb-and`. However, we found 16 sequences in `kernel32.dll` that follow the same pattern and could have been leveraged for a conditional branch gadget.

We then use the ADD/SUB gadget (see Table 5) to subtract `esi` from `eax` so that the latter holds the branch offset for `esp`. Finally, we move `eax` into `esp` using the stack as temporary storage. The STORE(eax) gadget (see Table 4) will store the branch offset on the stack, where `pop ebp` followed by the unconditional branch 1 gadget loads it into `esp`.

4.2 Extended Gadget Set

For those readers who have either written or analyzed real-world ROP exploits before, it would be clear to them that several other gadgets are useful in practice. For example, modern exploits usually invoke several WinAPI functions to perform malicious actions, e.g., launching

Type	Call-Preceded Sequence
Call 1	`lea eax,[ebp-34h]; push eax;`
	`call esi; ret`
Call 2	`call eax`
Call 3	`push eax; call [ebp+0Ch]`

Table 7: Function Call Gadgets

a malicious executable by invoking *WinExec()*. Calling such functions within a ROP attack requires a function call gadget (§4.2.1). It is also useful to have gadgets that allow one to conveniently write a NULL word to memory (the Null-Byte gadget) or the Stack-pivot gadget [17] which is used by attacks exploiting heap overflows. Our instantiations of the Null-Byte and Stack-pivot gadgets are given in the Appendix as they are not vital to understanding the discussion that follows.

Additionally, to provide a generic method for circumventing the behavioral heuristics of the Combined CFI Policy, we present a new gadget type, coined Long-NOP, containing long sequences of instructions which do not break the semantics of an arbitrary ROP chain (§4.2.2).

4.2.1 Call-Ret-Pair Gadget

CFI policies raise several challenges with respect to calling WinAPI functions within a ROP attack. First, one cannot simply exploit a `ret` instruction because the CFI_{RET} policy states that only a call-preceded sequence is allowed — clearly, the beginning of a function is not call-preceded. Second, the adversary must regain control when the function returns. Hence, the return address of the function to be called must point to a call-preceded sequence that allows the ROP attack to continue.

To overcome these restrictions, we utilize what we coined a *Call-Ret-Pair* gadget. The basic idea is to use a sequence that terminates in an indirect call but provides a short instruction sequence afterwards that terminates in a `ret` instruction. Among our possible choices, the Call 1 sequence shown in Table 7 was selected.

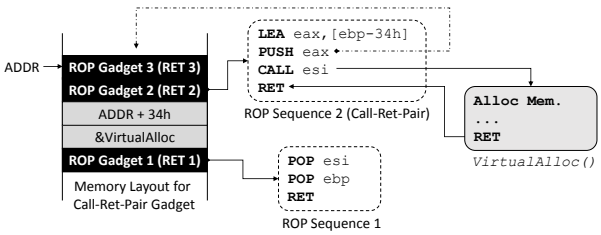


Figure 3: Example for Call-Ret-Pair Gadget

To better understand the intracies of this gadget, we provide an example in Figure 3. This example depicts how we can leverage our gadget to call *VirtualAlloc()*. We start with a load register gadget which first loads the start address of *VirtualAlloc()* into `esi`. Further, it loads into `ebp` an address denoted as `ADDR`. At this address is stored RET 3, the pointer to the ROP sequence we desire to call after *VirtualAlloc()* has returned. The next ROP sequence is our Call-Ret-Pair gadget, where the first instruction effectively loads RET 3 pointed to by `ebp-34h` into `eax`. Next, RET 3 is stored at `ADDR` onto the stack using a `push` instruction before the function call occurs. The `push` instruction also decrements the stack pointer so that it points to RET 2. The subsequent indirect call invokes *VirtualAlloc()* and automatically pushes the return address onto the stack, i.e, it will overwrite RET 2 with the return address. This ensures that the control-flow will be redirected to the `ret` instruction in our Call-Ret-Pair gadget when *VirtualAlloc()* returns. Lastly, the return will use RET 3 to invoke the next ROP sequence.

Note that this Call-Ret-Pair gadget works for subroutines following the *stdcall* calling convention. Such function remove their arguments from the stack upon function return. For functions using cdecl, we use a Call-Ret-Pair gadget that pops after the function call, the arguments of the subroutine from the stack. The details of the gadget we use for cdecl function can be found in the Appendix of our technical report [19].

For ROP attacks that terminate in a function call, we leverage the Call 2 and Call 3 gadgets in Table 7. The difference resides in the fact that Call 2 requires the target address to be loaded into `eax`, whereas Call 3 loads the branch address from memory.

Recall that the CFI policy for indirect calls (CFI_{CALL} in Table 2) only permits the use of branch addresses taken from an exported symbol or a valid code pointer place. However, as we already described in §3.2.4, the integrity of code pointers is not guaranteed. Hence, we can leverage GOT overwrite-like attacks to change the address at a given code pointer location. Alternatively, since modern applications typically make use of many WinAPI functions by default, we can indirectly call one of these functions using the external symbols.

4.2.2 Long-NOP Gadget

Our final gadget is needed to thwart the restriction that after $s = 7$ short sequences in a row is used, another sequence of at least $n = 20$ instructions must follow (see CFI_{HEU} in Table 2). For this task, we developed a new gadget type that we refer to as the long no-operation (long-NOP) gadget. Constructing long-NOP in a way that does not break the semantics of an arbitrary ROP chain was a non-trivial task that required painstaking analyses and a stroke of luck.

To identify possible sequences for this gadget type, we let our sequence finder filter those call-preceded sequences that contain more than $n = 20$ instructions. To ensure that the long sequence does not break the seman-

tics of the ROP chain, we further reduced the set of sequences to those that (i) contain many memory-write instructions, and (ii) make use of only a small set of registers. While the latter requirement is obvious, the former seems counter-intuitive as it can potentially change the memory state of the process. However, if we are able to control the destination address of these memory writes, we can write arbitrary values into the data area of the process outside the memory used by our ROP attack.

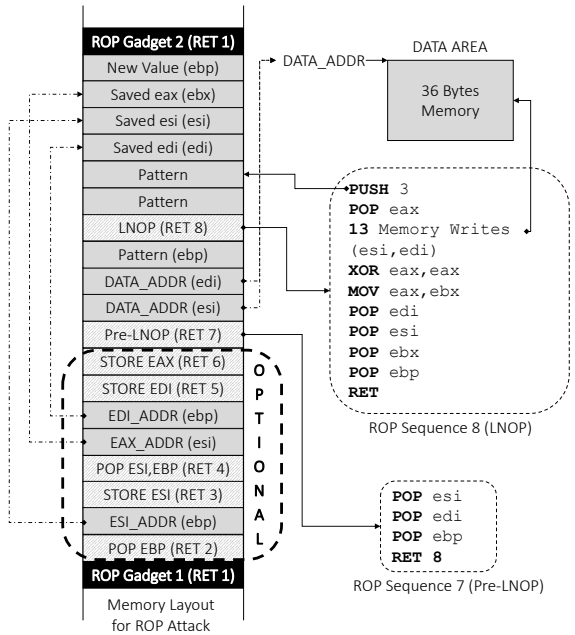


Figure 4: Flow of Long-NOP gadget

Among the sequences that fulfill these requirements, we chose a sequence that is (abstractly)[6] shown in Figure 4. It contains 13 memory write instructions using only the registers `esi` and `edi`. We stress that the entire gadget chain for long-NOP does not induce *any* side-effects, i.e., the content of all registers and memory area used by the ROP attack is preserved.

We distinguish between mandatory and optional sequences used for long-NOP. The latter sequences are only required if the content of all registers needs to be preserved. We classify them as optional, since it is very unlikely that ROP attacks need to operate on all registers during the entire ROP execution phase. If all registers need to be preserved (worst-case scenario), we require 6 ROP sequences before the long-NOP gadget sequence is invoked. Since all registers are preserved, we can issue in each round another ROP sequence until all desired ROP sequences have been executed.

Mandatory Sequences. The mandatory sequences are those labeled Sequence 7 and 8 (in Figure 4). Sequence 7 is used to set three registers: `esi`, `edi`, and `ebp`. We load in `esi` and `edi` the same address, namely `DATA_ADDR`, which points to an arbitrary data memory area in the address space of the application, e.g., stack, heap, or any other data segment of an executable module. Due to the `ret 8` instruction, the stack pointer will be incremented by 8 more bytes leaving space for pattern values. Afterwards, our long-NOP sequence uses `esi` and `edi` to issue 13 memory writes in a small window of 36 bytes. In each round, we use the same address for `DATA_ADDR`, and hence, we always write the same arbitrary values in a 36 byte memory space not affecting memory used by our ROP attack. The long-NOP sequence also destroys the value of `eax` and loads new values via `pop` instructions in other registers. However, these register changes are resolved by our optional sequences discussed next.

Optional Sequences. ROP Sequence 2 to 6 are the optional sequences, and are responsible for preserving the state of all registers. The optional sequences shown in Figure 4 represent those already presented in our basic gadget arsenal in §4.1. Depending on the specific goals and gadgets of a ROP attack, the adversary can choose among the optional sequences as required.

ROP Sequence 2 and 3 store the value of `esi` on the stack in such a way that the `pop esi` instruction in long-NOP resets the value accordingly. ROP Sequence 4 to 6 store the content of `eax` and `edi` on the stack. Similar to the store for `esi`, the content is again re-loaded into these registers via `pop` instructions at the end of the long-NOP sequence. However, the content of register `eax` and `ebx` is exchanged after the long-NOP sequence since `mov eax,ebx` stores `ebx` to `eax`, and the former value of `eax` is loaded via `pop` into `ebx`. However, we can compensate this switch by invoking the Long-NOP gadget twice so that `eax` and `ebx` are exchanged again.

5 Hardening Real-World Exploits

We now elaborate on the hardening of two real-world exploits against 32-bit Windows 7 SP1 and a Linux proof-of-concept exploit. Specifically, we transform publicly available ROP attacks against Adobe PDF reader [26] and the GNU mediaplayer mPlayer [10]. We used the gadget set derived in §4 to perform the transformation. Furthermore, our attacks are executed with the *Caller*, *SimExecFlow*, *StackPivot*, *LoadLib*, and *MemProt* option for ROP detection in Microsoft EMET 4.1 enabled. The source code for both attacks is given in our technical report [19].

5.1 Windows Exploits

The Adobe PDF attack used in this paper exploits the integer vulnerability CVE-2010-0188 in the TIFF image processing library `libtiff`. The vulnerability originally targeted Adobe PDF versions 9.1-9.3 running on Windows XP SP2/SP3. Likewise, the mPlayer attack ex-

ploited a buffer overflow vulnerability that allows the adversary to overwrite an exception handler pointer. Since we perform our analyses on Windows 7, we ported both exploits from Windows XP to Windows 7.

Exploit Requirements: For both exploits, we need to (1) allocate a new read-write-execute (RWX) memory page with *VirtualAlloc()*, (2) copy malicious shellcode into the newly allocated page by using *memcpy()*, and (3) redirect the control-flow to the shellcode. Originally, the exploits made use of non-call-preceded gadgets, and used a long chain of short instruction sequences. For mPlayer 18 consecutive short sequences are executed, while for Adobe PDF 11 sequences are executed until the first system call is issued. Hence, both exploits clearly violate CFI_{RET} and CFI_{HEU} of the combined CFI policy. These exploits are prevented by Microsoft EMET because of CFI_{RET}, and are detected by both kBouncer and ROPecker due to violation of the CFI_{HEU} policy.

Replacing ROP Sequences: A simplified view of the gadget chain we use for our hardened exploits in the PDF exploit is shown in Figure 5. We first replaced all non-call-preceded sequences with one of our call-preceded sequences in our ROP gadget set identified in Section 4. Both exploits mainly use load register and memory gadgets to set the arguments for *VirtualAlloc()* and *memcpy()*, and function call gadgets to invoke both functions. By leveraging only call-preceded sequences, our attacks comply to the CFI policy for returns (CFI_{RET}).

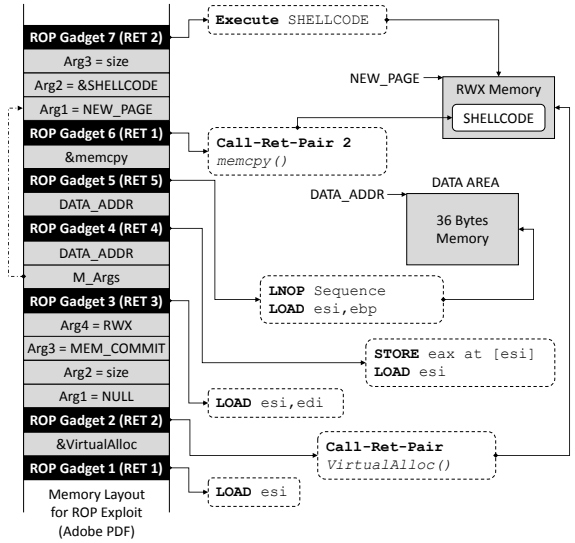


Figure 5: Simplified view of our hardened PDF exploit. See [19] for the full source code.

Since both exploits make use of WinAPI calls, we utilized our Call-Ret-Pair gadget to invoke *VirtualAlloc()* and *memcpy()*. As both functions are default routines used in a benign execution of Adobe PDF and mPlayer, we are allowed to leverage indirect calls to invoke these functions (addressing CFI_{CALL}). Note that even if this were not the case, we could still call these functions by overwriting valid code pointer locations. A demonstration of this weakness — particularly for the approach of Zhang and Sekar [46] — is provided in Section 5.2. Lastly, we need to tackle the CFI policies for behavioral heuristics (addressing CFI_{HEU}) by ensuring that we never execute more than 7 short sequences in a row before calling our long-NOP gadget.

Putting-It-All-Together: Gadget ❶ in Figure 5 loads the target address of *VirtualAlloc()* into `esi`. The arguments to this function (`Arg1-Arg4`) are set on the stack. They are chosen in such a way that *VirtualAlloc()* allocates a new RWX memory page. Gadget ❷ leverages our Call-Ret-Pair gadget to call *VirtualAlloc()*. The start address of the page is placed by *VirtualAlloc()* into `eax`.

ROP Gadgets ❸ and ❹ facilitate two goals: first they store the start address of the new RWX page on the stack. Second, they prepare the execution of the long-NOP gadget. In particular, they set `esi` and `edi` to DATA_ADDR. This address points to an arbitrary data section of one of the linked libraries. Our long-NOP sequence (ROP Gadget ❺) will later perform 13 memory writes on this data region, thereafter setting `esi` to the start address of *memcpy()*. ROP Gadget ❻ invokes *memcpy()* to copy the malicious shellcode onto the newly allocated RWX page. Lastly, our ROP chain transfers the control-flow to the copied shellcode via Gadget ❼, which in both exploits opens the Windows calculator.

For the Adobe PDF attack, we used 7 ROP sequences with 52 instructions executed. In the hardened version of the mPlayer exploit, we used 49 ROP sequences with 380 instructions executed. Note that the 49 sequences include the interspersed long-NOP sequences to adhere to the CFI policy CFI_{HEU}. We used a writable memory area of 36 Bytes for the long-NOP gadget. The requirement of more sequences for the mPlayer attack can be attributed to the fact that this exploit did not allow for the use of any NULL bytes in the payload and so we needed to leverage a NULL-Byte gadget (Appendix A) in this exploit. The mPlayer exploit also required a stack pivot gadget (Appendix B). This attack also required a specific stack pivot gadget adding a large constant to `esp`. Unfortunately, our stack pivot sequences in `kernel32.dll` did not use large enough constants, and the original sequence exploited a non call-preceded one in `avformat-52.dll`. However, we identified another useful call-preceded stack pivot sequence in the same library which allowed us to instantiate the exploit.

The above strategies can be used to easily transform other ROP attacks to bypass current coarse-grained CFI defenses. Furthermore, given our routines for finding and filtering useful call-preceded ROP sequences, the process

of transforming exploits could be fully automated. We leave that as an exercise for future work.

A final remark concerns the control transfer to the injected shellcode. In both exploits, we invoke a call-preceded sequence terminating in an indirect jump. While this approach works for kBouncer, ROPecker, and ROPGuard, it might raise an alarm for CFI for COTS binaries if the shellcode is placed at an address that is not within the set of valid function pointers (i.e., indirect jump targets). However, there are several ways to tackle this issue. A very effective approach has been shown by Göktas et al. [22], where the code section is simply set to be writable, the shellcode copied to an address which resembles a valid function pointer, and after which the code section is reset back to be executable. Alternatively, one can overwrite the location of a valid function pointer with the start address of the shellcode. We provide a detailed example how this can be realized in the next subsection.

5.2 Linux Shellcode Exploit

Since the approach of Zhang and Sekar [46] targets Linux specifically, we also developed a proof-of-concept exploit that shows how our attack bypasses the CFI policies for indirect calls. To do so, we use a sample program that suffers from a buffer overflow vulnerability allowing an adversary to overwrite a return address on the stack. The goal of our attack is to call *execve()*, which is a standard system function defined in `libc.so` to execute a new program. The challenge, however, is that the example program does not include *execve()* in its external symbols, and consequently, we are not allowed to redirect the control-flow to *execve()* using an indirect call.

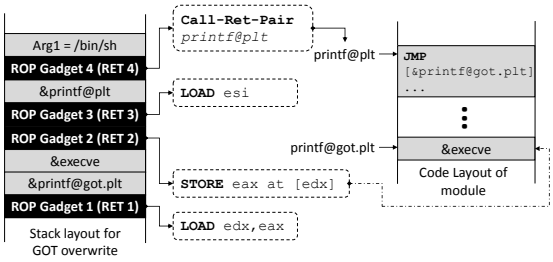


Figure 6: GOT overwrite attack

To overcome this restriction, we make use of an old (but seemingly forgotten) attack technique called global offset table (GOT) overwrite [9]. The basic idea is to write the address of *execve()* at a valid code pointer location. A well-known location for doing so is the GOT table, which contains pointers to library calls such as *printf()*. We reiterate that the weakness here is that CFI for COTS binaries does not validate the integrity of these pointers — a very difficult, if not unsurmountable task, in the current design of Linux since the GOT is initialized at runtime of an application. Hence, we can invoke gadgets to overwrite the pointers placed in the GOT. Specifically, we first find useful sequences from the Linux standard library `libc.so` and use gadgets that perform the GOT overwrite while using only call-preceded sequences.

Putting-It-All-Together: An example on how we bypass the CFI policy for indirect calls is shown in Figure 6. The approach is as follows: first, Gadget ❶ loads the address of the GOT entry we want to modify into `edx`, and loads `eax` with the address of *execve()*. Next, Gadget ❷ overwrites the address of *printf()* with the address of *execve()* in the GOT. Finally, Gadget ❸ loads the address of the *printf()* stub into `esi`, and Gadget ❹ uses a Call-Ret-Pair gadget to invoke *execve()*. At this point, the attack succeeds without violating any of the CFI policies.

5.3 On Parameter Adjustment

As alluded to in §3.3, adjusting the parameters for the CFI_{HEU} policy beyond the recommended settings will negatively impact the false positive rate. To assess that, we extended the analysis beyond what Pappas et al. [31] originally performed in order to analyze the impact of increasing n to 30 or 40 instructions — thereby rendering our Long-NOP gadget (which is only 23 instructions long) stitching ineffective. Specifically, we performed an experiment using three benchmarks of the SPEC CPU 2006 benchmark suite: `bzip2`, `perlbench`, and `xalancbmk`. The first two are programmed in C, while the latter in C++. We developed an Intel Pintool that counts the number of instructions issued between two indirect branches, and the number of consecutive short instruction sequences. Whenever a function call occurs, we check how many short sequences (s) have been executed since the last function call.

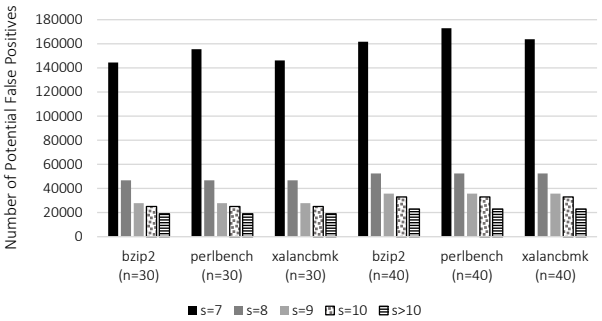


Figure 7: Potential false positives when the parameters for the consecutive sequences (s) and sequence length (n) are adjusted.

As Figure 7 shows, increasing the thresholds for n induces many potential false positives (y-axis). In particular, for each benchmark (x-axis), observe that for $s > 10$ there are about 20,000 potential false positives, i.e., 20,000 times we detected a function call that was preceded by more than 10 short sequences[7].

6 Related Work

Concurrent and independent to our work, several research groups have investigated the security of coarse-grained CFI solutions [11, 22–24, 34]. However, our analysis differs from these works as we examine the security of a combination of coarse-grained CFI policies irrespective of when the CFI check occurs. For instance, the attacks shown in [11, 22, 34] are prevented by our combined CFI policy which monitors the sequence length at any time in program execution. Furthermore, unlike these works, we systematically show the construction of a Turing-complete gadget set based on a weak adversary that has only access to one standard shared Windows library. On the other hand, concurrent work also investigates some other interesting attack aspects: Göktas et al. [22] demonstrate attacks against CCFIR [45] using call-preceded gadgets to invoke sensitive functions via direct calls; Carlini and Wagner [11] and Schuster et al. [34] show flushing attacks that eliminate return-oriented programming traces before a critical function is invoked.

Lastly, new CFI-based solutions have also been proposed. For instance, the approaches of Tice et al. [40] and Jang et al. [25] focus on protecting indirect calls to virtual methods in C++. Both approaches have been implemented as a compiler extension and ensure that an adversary cannot manipulate a virtual table (vtable) pointer so that it points to an adversary-controlled (malicious) vtable. Unfortunately, these schemes do not protect against classical ROP attacks which exploit return instructions, and map malicious code to a memory area reserved for a valid virtual method.

7 Summary

Without question, control-flow integrity offers a strong defense against runtime attacks. Its promise lies in the fact that it provides a general defense mechanism to thwart such attacks. Rather than focusing on patching program vulnerabilities one by one, CFI's power stems from focusing on the integrity of the program's control flow regardless of how many bugs and errors it may suffer from. Unfortunately, several pragmatic issues (most notably, its relatively high performance overhead), have limited its widespread adoption.

To better tackle the performance trade-off between security and performance, several coarse-grained CFI solutions have been proposed to date [13, 20, 31, 45, 46]. Additionally, it has been recently shown that such coarse-grained CFI policies can be applied to operating system kernels [16]. These proposals all use relaxed policies, e.g., allowing returns to target any instruction following a call instruction.

While many advancements have been made along the way, all to often the relaxed enforcement policies significantly diminish the security afforded by Abadi et al. [3]'s seminal work. This realization is a bit troubling, and calls for a broader acceptance that we should not sacrifice security for small performance gains. Doing so simply does not raise the bar high enough to deter skillful adversaries. Indeed, our own work shows that even if coarse-grained CFI solutions are combined, there is still enough leeway to mount reasonable and Turing-complete ROP attacks. Our hope is that our findings will raise better awareness of some of the critical issues when designing robust CFI mechanisms, all-the-while re-energizing the community to explore more efficient solutions for empowering CFI.

8 Acknowledgments

We thank Kevin Z. Snow and Úlfar Erlingsson for their valuable feedback on earlier versions of this paper.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, G. C. Necula, and M. Vrable. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13 (1), 2009.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 2000.

[5] E. Bachaalany. Inside EMET 4.0. REcon Montreal, 2013. Presentation. Slides: `http://recon.cx/2013/slides/Recon2013-Elias%20Bachaalany-Inside%20EMET%204.pdf`.

[6] blexim. Basic integer overflows. *Phrack Magazine*, 60(10), 2002.

[7] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[8] M. Budiu, U. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 2006.

[9] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 56(5), 1996.

[10] C4SS!0 and h1ch4m. MPlayer Lite r33064 m3u Buffer Overflow Exploit (DEP Bypass). `http://www.exploit-db.com/exploits/17565/`, 2011.

[11] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[13] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[14] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[15] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[16] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy*, Oakland '14, 2014.

[17] D. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010. Presentation. Slides: `http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`.

[18] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[19] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. Technical Report TUD-CS-2014-0097, Technische Universität Darmstadt, 2014.

[20] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. `http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`, 2012.

[21] gera. Advances in format string exploitation. *Phrack Magazine*, 59(12), 2002.

[22] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, Oakland '14, 2014.

[23] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[24] S. Jalayeri. Bypassing EMET 3.5's ROP mitigations. `https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/`, 2012.

[25] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[26] jduck. The latest Adobe exploit and session upgrading. `http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html`, 2010.

[27] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University Bochum, 2009.

[28] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[29] Microsoft. Enhanced Mitigation Experience Toolkit. `https://www.microsoft.com/emet`, 2014.

[30] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), 2001.

[31] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[32] J. Pewny and T. Holz. Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[33] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.

[34] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2014.

[35] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[36] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, Oakland '13, 2013.

[37] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[38] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista. `http://www.phreedom.org/research/bypassing-browser-memory-protections/`, 2008.

[39] M. Thomlinson. Announcing the BlueHat Prize winners. `https://blogs.technet.com/b/msrc/archive/2012/07/26/announcing-the-bluehat-prize-winners.aspx?Redirected=true`, 2012.

[40] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[41] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Symposium on Research in Attacks, Intrustions, and Defenses (RAID)*, 2012.

[42] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[43] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[44] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[45] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy*, Oakland '13, 2013.

[46] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

A NULL-Byte Write Gadget

In real-world exploits it is useful to have gadgets that allow one to conveniently write a NULL word to memory. This is important as real-world vulnerabilities typically do not allow an adversary to write a NULL byte in the payload, but such functionality is indeed needed to write a 32-bit NULL word on the stack when required as a parameter to function calls.

A prominent example is the traditional *strcpy(dest,src)* vulnerability, which can be exploited to write data beyond the boundaries of the *src* variable. However, *strcpy()* stops copying input data after encountering a NULL byte.

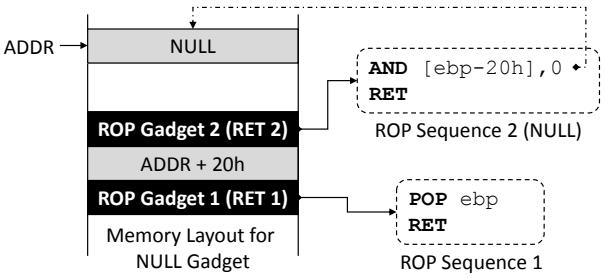


Figure 8: Details of NULL Gadget

Our choice for such a gadget is shown in Figure 8. This gadget first loads the target address into `ebp` with the first ROP sequence. The next sequence exploits the `and` instruction to generate a NULL word at the memory location pointed to by `ebp-20h`.

B Stack Pivot Gadgets

We take advantage of two distinct stack pivot gadgets shown in Table 8. The first one is our unconditional branch gadget, which moves `ebp` via the leave instruction to `esp`. The other sequence takes the value of `esi` and loads it into `esp`. In both sequences, the adversary must control the source register `ebp` and `esi`, respectively. This is achieved by invoking a load register gadget beforehand. Note also that several vulnerabilities allow an adversary to load these registers with the correct values at the time the buffer overflow occurs, which would make the ROP attack easier.

Type	Call-Preceded Sequence (ending in `ret`)
Pivot 1	`leave`
Pivot 2	`mov esp, esi; pop ebx; pop edi; pop esi; pop ebp`

Table 8: Stack Pivot Gadgets

C Details of Long-NOP Gadget

```
pop esi ; ptr to writable mem for NOP
pop edi ; ptr to writable mem for NOP
pop ebp ; unused in NOP
retn 8  ; -> insert 8 bytes junk after
    next gadget
```

Listing 1: Pre-Seuence for LNOP

```
movzx eax, ax
mov [esi+4], eax     ; 5 writes to
mov [esi+8], 1F4Bh   ; a 20 byte
mov [esi+14h], 5     ; memory region
mov [esi+10h], 1Fh
mov [esi+0Ch], 0Ch
push 3Bh
pop eax
mov [esi+1Ch], eax   ; 2 writes to
mov [esi+20h], eax   ; 8 byte region
xor eax, eax
mov [esi+18h], 17h   ; another 8 bytes
mov [esi+24h], 98967Fh
mov [edi+18h], eax   ; if edi == esi
mov [edi+1Ch], eax   ; these writes
mov [edi+20h], eax   ; goto the same
mov [edi+24h], eax   ; region as before
pop edi    ; (optional:) restore edi
pop esi    ; (optional:) restore esi
mov eax, ebx
pop ebx    ; (optional:) load former eax
pop ebp
retn 0Ch
```

Listing 2: Long sequence used for LNOP gadget

Notes

[1]Some of the mechanisms used in kBouncer and ROPGuard (both awarded by Microsoft's BlueHat Prize [39]) have already been integrated in Microsoft's defense tool called EMET [29].

[2]Sequences that end in indirect jumps or calls can also be used [12].

[3]Typically, CFI does not validate direct branches because these addresses are hard-coded in the code of an executable and cannot be changed by an adversary when W⊕X is enforced.

[4]Specifically, kBouncer reports a ROP attack when a chain of 8 short sequences has been executed, where a sequence is referred to as "short" whenever the sequence length is less than 20 instructions.

[5]The target address of an external function is dynamically allocated in the global offset table (GOT) which is loaded by an indirect memory jump in the procedure linkage table (PLT).

[6]For the interested reader, we have placed the specific assembler implementation of the long-NOP sequence in Appendix C.

[7]We also simulated the analysis performed in [31] by setting $n = 20$. However, we arrive at a significantly higher false positive rate than in [31]. This is likely due to the fact that we perform our analysis on industry benchmark programs, while their analysis is based on opening web-browsers or document readers. Furthermore, their focus is on WinAPI calls, whereas in Figure 7 we instrument every call.

Size Does Matter

Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard

Enes Göktaş
Vrije Universiteit
Amsterdam, The Netherlands

Elias Athanasopoulos
FORTH-ICS
Heraklion, Crete, Greece

Michalis Polychronakis
Columbia University
New York, NY, USA

Herbert Bos
Vrije Universiteit
Amsterdam, The Netherlands

Georgios Portokalidis
Stevens Institute of Technology
Hoboken, NJ, USA

Abstract

Code-reuse attacks based on return oriented programming are among the most popular exploitation techniques used by attackers today. Few practical defenses are able to stop such attacks on arbitrary binaries without access to source code. A notable exception are the techniques that employ new hardware, such as Intel's Last Branch Record (LBR) registers, to track all indirect branches and raise an alert when a sensitive system call is reached by means of *too many* indirect branches to *short* gadgets—under the assumption that such gadget chains would be indicative of a ROP attack. In this paper, we evaluate the implications. What is "too many" and how short is "short"? Getting the thresholds wrong has serious consequences. In this paper, we show by means of an attack on Internet Explorer that while current defenses based on these techniques raise the bar for exploitation, they can be bypassed. Conversely, tuning the thresholds to make the defenses more aggressive, may flag legitimate program behavior as an attack. We analyze the problem in detail and show that determining the right values is difficult.

1 Introduction

Modern protection mechanisms like data execution protection (DEP) [2], address space layout randomization (ASLR) [26] and stack smashing protection (SSP) [9] are now available on most general-purpose operating systems. As a result, exploitation by injecting and executing shellcode directly in the victim process has become rare. Unfortunately, these defenses are not sufficient to stop more sophisticated attacks.

Nowadays, attackers typically use memory disclosures to find exactly the addresses ASLR is trying to hide [30, 34, 36]. Likewise, there is no shortage of tutorials on how to evade state-of-the-art defenses [14, 28]. Attackers are able to hijack control flow and bypass DEP by reusing code that is already available in the binary itself, or in the libraries linked to it. There are several variations of this exploitation method: return-to-libc [37], return-oriented programming (ROP) [31], jump-oriented programming [3, 6], and sigreturn oriented programming (SROP) [4]. Code reuse attacks, and especially ROP, may be the most popular exploitation method used by attackers today, bypassing all popular defense mechanisms. Even additional and explicit protection against ROP attacks over and beyond DEP, ASLR and SSP, such as provided by Microsoft's Enhanced Mitigation Experience Toolkit (EMET), do not stop the attacks in practice [14].

ROP attacks start when an attacker gains control of the stack and diverts the control to a *gadget*: a short sequence of instructions that performs a small subset of the desired functionality and ends with a `ret` instruction. Since the attackers control the return addresses on the stack, they can make the `ret` of one gadget jump to the start of another gadget, daisy chaining the desired functionality out of a large set of small gadgets.

It is no wonder, then, that the security community has scrambled to find alternative methods to defend software assets. For instance, over the past decade or so, there has been a tremendous amount of research interest in control flow integrity (CFI) [1]—a technique to prevent *any* flow of control not intended by the original program. Unfortunately, CFI is fairly expensive. Moreover, research has shown that attempts to make it faster and more practical by employing looser notions of integrity, make it vulnerable to exploitation again [16].

KBouncer and friends Perhaps the main and most practical defense mechanism proposed against ROP attacks nowadays is the one pioneered by kBouncer [25]—grand winner of the Microsoft Blue Hat Prize in 2012. The technique has become quite successful and despite its recent pedigree, it is already used in commercial products like HitmanPro's new Alert 3 service [18].

KBouncer and related approaches like ROPecker [8] use a new set of registers, known as the Last Branch Record (LBR), available in modern Intel CPUs. The registers can be used to log the last n indirect branches taken by the program. Using the LBR, kBouncer checks whether the path that lead to a sensitive system call (like `VirtualProtect`) contains "too many" indirect branches to "short" gadgets—which would be indicative of a ROP chain.

The two obvious questions that we need to ask are: what is "too many," and how short is "short"?

Specifically, suppose the defensive mechanism has thresholds T_C and T_G, such that it raises an alarm when it sees a chain of T_C or more gadgets of at most T_G instructions each. If the attackers can find just a single gadget greater than T_G that they can simply squeeze in between the others to break the sequence, the defensive mechanism would not detect it. Conversely (and more worryingly), if a program itself exhibits T_C gadgets of at most T_G instructions during normal execution, the defense mechanism would erroneously flag it as an attack.

Implemented carefully, the protection offered by this method is quite powerful, but picking the right values for T_G and T_C is a delicate matter. After all, the former scenario suggests that T_G is too small. However, incrementing T_G may lead to more false positives (FPs) because benign execution paths are more likely to contain T_C such gadgets.

Contributions In this paper, we investigate the problem of picking the right values for these two thresholds. We also evaluate whether the solutions proposed today are sufficient to stop exploitation in real software. Specifically, we show that while they raise the bar for exploitation significantly, they can be bypassed. As a demonstration, we discuss a proof of concept exploit against Internet Explorer that bypasses current kBouncer-based defenses. We then analyze the problem by considering the availability of gadgets of different lengths and determining the sequences of gadgets en route to sensitive system calls.

While this work does not fully explore the possibility of FPs with the thresholds used in literature, it shows that defining restrictive thresholds, which do not allow the composition of ROP payloads, is extremely complicated and may not be possible for many applications due to FPs. Finally, we discuss various avenues for ameliorating these techniques and provide evidence that setting the thresholds based on the application at-hand can significantly encumber attackers.

Outline The remainder of this paper is organized as follows. Section 2 provides some background information regarding code-reuse attacks and defenses that use gadget-chain length for detection. Section 3 discusses the weaknesses of such approaches, and in Sec. 4 we present the process of creating an exploit that can circumvent them. We propose countermeasures and discuss possible obstacles for their adoption in Sec. 5. In Sec. 6, we present the results of our experiments that indicate that one of the proposed countermeasure can improve detection. Related work is in Sec. 7 and we conclude in Sec. 8.

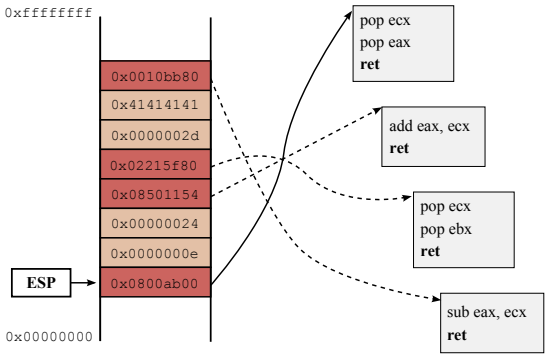


Figure 1: A very simple ROP chain that calculates $0xe + 0x24 - 0x2d$. Result is in the `eax` register.

2 Background

2.1 ROP and Code-reuse Attacks

ROP attacks are the most common vector for launching code-reuse attacks and require that the attacker gains control of the program's stack. By corrupting the return address of the executing function, upon its return, control is diverted to a *gadget* of the attacker's choosing. Gadgets are small sequences of code that end with a `ret`. By carefully positioning data on the stack, the attacker can make the program jump from one gadget to another, chaining together pieces of already existing code that implement the desirable payload, as shown in Fig. 1. While the gadgets that the attacker chains together are usually short in length (i.e., in number of instructions) and limited in functionality, previous work has shown that the attack is Turing complete [31]. That is, applications contain enough gadgets to perform arbitrary computations.

Creating a working ROP exploit is often a complex, multi-step process. It typically starts with a memory disclosure that allows the attacker to obtain code pointers. Next, the attack may require a variety of further preparations, such as advanced heap feng shui [35] to pave the way for a dangling pointer exploit, stack pivoting, and/or buffer overflows. In addition, the attacker needs to identify useful gadgets and construct a ROP program out of them by setting up the appropriate addresses and

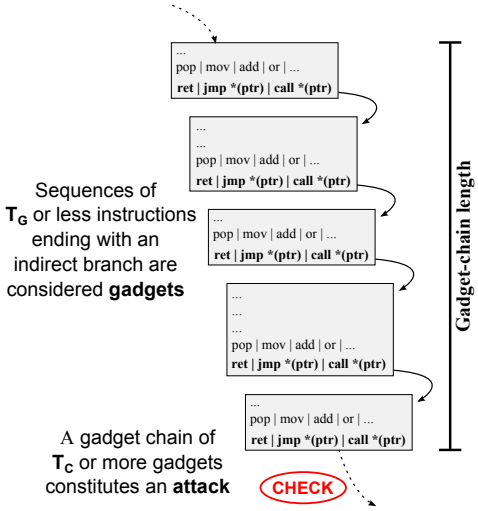


Figure 2: Example of a gadget chaining pattern used to identify code-reuse attacks.

arguments on the (possibly new) stack. Finally, a control flow diversion should start off the ROP chain.

ROP is popular despite its complexity because it provides a way for attackers to bypass defenses like DEP [2]. As a result, many recent works have focused on preventing ROP and other forms of code-reuse attacks [7, 8, 13, 23–25, 31, 33]. Other works have shown that a similar attack can also be performed with gadgets that end with indirect jump or call instructions instead of returns [3, 6, 21].

2.2 Monitoring Gadget Chains to Detect Attacks

kBouncer [25] and ROPecker [8] are two of the most easy to deploy solutions to stop ROP-like attacks. They employ a recent feature of Intel CPUs, known as the Last Branch Record (LBR), that logs the last branches taken by a program in a new set of registers [20, Sec. 17.4]. Intel introduced LBR for both the x86 and x86-64 architectures, so that, with the right configuration, the operating system (OS) is able to log the targets of indirect branches (including calls, jumps, and returns) in 16 machine-specific registers (MSR) registers with little overhead. These registers are accessible only from the OS kernel and are continuously overwritten as new branches occur.

A key observation for detecting ROP attacks, in both kBouncer and ROPecker, is that the attacks need to chain together a significant number of small gadgets to perform any useful functionality, like in the example shown in Fig. 2. From a high-level perspective, they include two parameters: the first controls what is the longest sequence of instructions ending with an indirect branch that will be considered a gadget, and the second specifies the number of successively chained gadgets that indicates an attack. We will refer to these two thresholds as T_G and T_C. These two parameters control the level of difficulty for performing an attack under these solutions. Increasing T_G or reducing T_C makes the construction of ROP payloads harder. However, overdoing it can lead to false positives (FP), due to legitimate execution paths being misclassified as attacks at run time. In the remainder of this section, we will briefly highlight kBouncer and ROPecker.

2.2.1 kBouncer

kBouncer kicks in every time a sensitive API call, like `VirtualProtect()`, `CreateProcess()`, etc., is executed by inserting hooks through the Detours [19] framework for Windows. It then scans the LBR registers to detect if the API call was made by a malicious ROP gadget chain, and terminates the running process if it was.

Two mechanisms are used to determine if there is an attack. The first mechanism aims to identify abnormal function returns. It is based on the observation that ROP chains manipulate control-flow to redirect control to arbitrary points in the program, where the attacker-selected gadgets reside. This constitutes a deviation from legitimate behavior, where returns transfer control to instructions immediately following a call. kBouncer checks the targets of all return instructions in the LBR to ensure that they are preceded by a call instruction. In x86 architectures where unaligned instructions are permissible, this call instruction does not necessarily need to be one actually intended by the program and emitted by the compiler. Any executable byte with the value of *0xE8*, one of the opcodes for the `call` instruction, can be actually considered as an *unintended* call instruction and attackers can use the gadget following it.

Recently, even just using gadgets following *intended* calls was shown to be sufficient to compose ROP payloads [16]. In anticipation of the possibility of such attacks, kBouncer introduced a second mechanism, based on gadget-chain length, to detect and prevent attacks. First, all potential gadgets are identified through offline analysis of an application. Every uninterrupted sequence of at most *20 instructions* ending in an indirect branch is treated as a potential gadget. At run time, kBouncer checks that there is no uninterrupted chain of *eight* or more such gadgets as targets in the LBR. In this case, the maximum gadget length of $T_G = 20$ was selected arbitrarily [25, Sec. 3.2], while through experimentation with a set of Windows applications, it was determined that a safe choice for the gadget-chain length threshold is $T_C = 8$ [25, Sec. 3.2, Fig. 7].

2.2.2 ROPecker

Similarly to kBouncer, ROPecker [8] utilizes LBR to detect ROP attacks. However, instead of only checking LBR registers upon entry to sensitive API calls, it introduces a new mechanism for triggering checks more often. It maintains a sliding window of code that is executable, while all other code pages are marked as non-executable. Checks are made each time a permission fault is triggered because control flow is transferred outside the sliding window. The intuition behind this approach is that due to code locality page faults are not triggered very often and ROP attacks are unlikely to use only gadgets contained within the sliding window (between 8 and 16 KB), so a check will be triggered before the attack completes.

Attack detection occurs primarily by checking gadget-chain length, like in kBouncer. However, ROPecker also checks for attacks in future returns by inspecting the return addresses stored in the stack. Potential gadgets are collected offline by statically analyzing applications, but they are defined differently from kBouncer. In particular, a gadget is a sequence of no more than *six instructions* that ends with an indirect branch, but does *not* contain any direct branches. Experiments were conducted with various Linux applications and benchmarks to determine a safe choice for the gadget-chain length threshold that will indicate an attack. The results varied, but a chain of *at least 11* gadgets was determined to be a safe choice. However, using a per-application threshold, if possible, is recommended. To summarize, the maximum length of a gadget is set to $T_G = 6$ and is selected arbitrarily [8, Sec. VII.B], while the safe choice for the gadget-chain length threshold is $T_C = 11$ [8, Sec. VII.A].

In the presence of multiple smaller gadget chains, intentionally created by mixing long and short gadgets to evade the mechanism, ROPecker also proposes accumulating the lengths of the smaller chains across multiple windows and using that instead, to gain a certain degree of tolerance to such attacks. Accumulation is done every three windows, and experimental results showed that an acceptable threshold for cumulative gadget-chain length is $T_{CC} = 14$.

3 The Problem

Both systems we study in this paper heavily depend on two parameters, namely T_C (the chain length) and T_G (the gadget length). In this section, we discuss the problem of picking the right values for T_C and T_G, and the way attackers can bypass the defenses proposed by kBouncer, ROPecker, and similar approaches.

What Is the Right Size? Mechanisms like kBouncer and ROPecker rely on defining gadgets based on the size of instruction sequences ending in indirect branches, and detect attacks based on the size of gadget chains. The problem with such measures is that while they do raise the bar, they are also their own Achilles' heel. By interspersing their ROP code with an occasional longer sequence of instructions (ending with an indirect branch) that will not be registered as a gadget, an attacker can reduce the length of gadget chains, as observed by these systems, and avoid detection.

Figure 3 shows a high-level overview of such an attack. After receiving control through an exploit, an attacker first uses up to $T_C - 1$ *detectable gadgets* (DG). Then, he employs at least one longer *undetectable gadget* (UG), that is, a sequence of more than T_G instructions. To be precise, an attacker may need to use an UG earlier for the first time because a chain of T_C legitimate application gadgets may already exist before he receives control, leading to a longer chain of DGs. If a check is triggered while this chain is still visible in the LBR, the attack will be detected. kBouncer only conducts checks on certain API calls, so an attacker needs to only worry about the number of DGs in the LBR when performing such calls. On the other hand, ROPecker triggers checks more frequently, but, exactly due to this fact, uses less restrictive T_C and T_G parameters. In the worst case, an attacker needs to use an UG first in his ROP chain.

Weak Control-Flow Enforcement kBouncer performs an additional check to ensure that the targets of all return instructions in the LBR point to instructions preceded by calls. However, recent work [16] has shown that it is possible to build a ROP payload using such call-preceded (CP) gadgets and evade even stricter control-flow restrictions. As a result, the effectiveness of defenses like kBouncer depends entirely on the T_C and T_G parameters. In the example shown in Fig. 2, the attacker would not be able to link gadgets that are not preceded by calls using function returns.

Accumulating Gadget-Chain Lengths ROPecker proposes an extension to tackle exactly the problem of mixing long and short gadgets. They define another parameter, T_{CC}, which is the threshold for the cumulative length of gadget chains in three successive windows and, hence, checks. This extension aims to prevent attacks following the pattern shown in Fig. 3. However, ROPecker does not consider instruction sequences including direct branches as gadgets, so an attacker can employ those as alternative shorter UGs. Furthermore, attackers can carefully construct attacks that consist of a small number of gadgets and then inject code, as it was

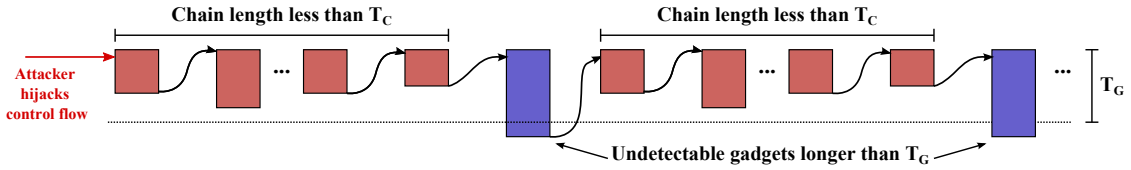


Figure 3: Mixing shorter and longer gadgets to avoid detection. Gadgets larger than T_G instructions are not considered as gadgets by both kBouncer and ROPecker. The latter also ignores gadgets that contain direct branch instructions.

done in previous work [16]. Details follow in the next section.

4 Proof-of-Concept Attack

In this section, we describe the construction of a proof-of-concept (PoC) exploit, which can compromise a vulnerable binary running under kBouncer. We have selected kBouncer, since we consider it the hardest to evade of the two systems examined in this paper. Recall that kBouncer is based on restricting ret instructions, so that they can only redirect control flow to gadgets preceded by an intended or unintended call instruction, and on a heuristic that scans for long chains of consecutive gadgets, as they are defined by kBouncer. The constructed exploit is generic, because it uses gadgets solely from the `shell32.dll` library which is shared among many widely used applications in Windows, and it is also effective against similar approaches, like ROPecker [8].

Previous work [16] has already shown that it is possible to compose attacks using an even more limited set of gadgets, that is, only gadgets following intended call instructions and starting at function entry points. We build on this prior knowledge to collect the gadgets that are available under kBouncer and show that we can build an exploit that remains undetectable. More importantly, we show that we can construct a very short payload that could not be easily detected unless T_C and T_G are set to considerably more restrictive values.

4.1 Preparation

The vulnerability we use to build our exploit is based on a real heap overflow in Internet Explorer [27] and has been also used in multiple other works [16,34] in the past. The first part of the exploit deals with disclosing information to bypass ASLR and then controlling the target address of an indirect jump instruction. Details of the preparation phase can also be found in previous work [16]. Here, we summarize the initial steps that are common with previous work and introduce new actions that are necessary for completing this exploit.

The vulnerability is triggered by accessing the `span` and `width` attributes of an HTML table's column through JavaScript. A great feature of the vulnerability is that it can be triggered repeatedly to achieve different tasks. First, it can be triggered to overwrite the size attribute of a string object, which consequently allows the `substring()` method of the string class to read data beyond the boundary of the string object, as long as we know the relative offset of that data from the string object. The `substring()` method serves as a memory disclosure interface for us. Second, it can be triggered to overwrite the virtual function table (VFT) pointer within a button object. Later, when we access the button object from within carefully prepared JavaScript code, the program will operate on the overwritten data and will eventually grant us control over an indirect jump instruction.

Due to ASLR being in use, we need to exploit the string object to "learn" where `shell32.dll` is loaded at run time, i.e., its base address. Before anything else, we use heap Feng Shui [35] to position the vulnerable buffer, and the string and button objects in the right order, so that we can overflow in the string object without concurrently receiving control of the indirect jump. The following steps are taken to locate `shell32.dll`, the first two steps have been also part of prior work, while the latter was added to achieve our end goal:

1. This vulnerability allows us to easily locate `mshtml.dll`. The button object's VFT contains a pointer to a fixed offset within the DLL. After heap Feng Shui, the button object follows the string object in memory at a fixed distance, so we use the controlled string object to read that pointer and reveal the location of the DLL. `mshtml.dll` contains pointers directly to `shell32.dll`, however, they are located in its *Delayed Import Address Table* (IAT), so they are not available at the time of exploitation.

2. In contrast to `mshtml.dll`, `ieframe.dll` do contain pointers to `shell32.dll` in its normal IAT, which gets loaded during the initiation of libraries. So `ieframe.dll` has the pointers to `shell32.dll` we are looking for available at the time of exploitation. As a result, by learning the base address of `ieframe.dll`, we can achieve our end goal. `mshtml.dll` has pointers

to `ieframe.dll` available in its Delayed IAT, but to read that, we first need to calculate its relative offset from the string. Since we already know the base address of `mshtml.dll`, we just need to find out the address of the string object, so we can calculate offsets within the DLL. Fortunately, the button object contains an address that has a constant distance from the beginning of the string object, so by first exfiltrating that, we can calculate the base address of `ieframe.dll`.

3. Since we now know the base address of `ieframe.dll`, we exploit the string object once more to read a pointer to `shell32.dll`, thus revealing its base address. `shell32.dll` also allows us to locate `VirtualProtect()` through its own IAT.

Finally, we need to also determine the location of a buffer we control, which we use to store the ROP payload, shellcode, etc. We use heap spraying [11] to create many copies of such a buffer in the process' memory, which has the effect of placing one of the copies at an address that can be reliably determined. Heap spraying is not foolproof, however, it works consistently in this particular case.

4.2 Collecting Gadgets

When kBouncer and friends are active the `ret` instruction can only target gadgets that are preceded by a call instruction. Previous work has referred to such gadgets as call-site (CS) gadgets [16], we will use this term to refer to them. CS gadgets are a subset of all the gadgets available in traditional ROP and JOP attacks, and include gadgets defined by intended and unintended call instructions. So any bytes in the program that could be interpreted as a call instruction, subsequently introduce a CS gadget. Note that the entire set of gadgets is still present in the application, but it can now only be targeted by indirect jump and call instructions. Generally, we will use the CS prefix with gadgets that are call preceded and the type of indirect branch ending the gadget as suffix (e.g., `RET` or `CALL *`).

To find usable gadgets, we disassemble the target binary multiple times. We start disassembling from each individual byte in the code segment of each image, until we encounter a stop condition, which can be an indirect control flow transfer, or an invalid or privileged instruction. This is similar to the static analysis phase of kBouncer that determines the locations of gadgets. Like kBouncer, we follow direct branches and calculate the length of a gadget using the shortest number of instructions that can execute from the beginning of the gadget till an indirect branch. This means that in the presence of conditional branches, we follow both paths and use the shortest one as the length of the gadget.

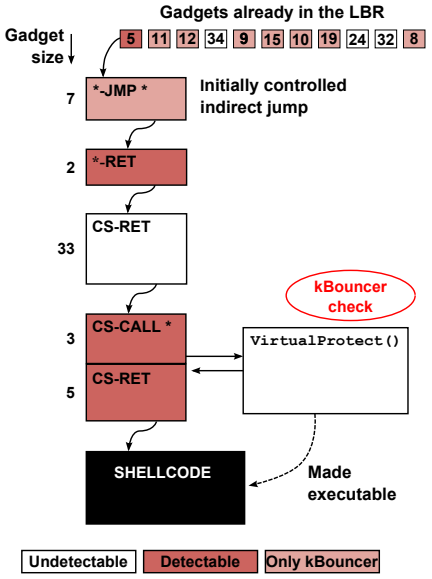


Figure 4: PoC exploit that bypasses both kBouncer and ROPecker. This figure focuses more on the details related with kBouncer, since it uses stricter detection thresholds. Gadgets receiving control through a return are all call-preceded, depicted using the *CS* prefix in the figure. The exploit uses one heuristic breaking gadget to keep the chain of detectable gadgets small and calls `VirtualProtect()` which triggers a kBouncer check. Note that kBouncer and ROPecker fail to detect gadgets longer than 20 and 6 instructions, respectively.

4.3 Heuristic Breakers

Since our exploit should fly under kBouncer's defensive radar, we have to ensure that we do not use sequences of more than seven short gadgets at any time (i.e., kBouncer-gadgets with 20 instructions or less). We avoid doing so by using a long gadget that performs minimal work (i.e., only sets a single register) as part of the exploit. Generally, to avoid the detection an attacker needs to intersperse the ROP chain with long gadgets. We call such gadgets *heuristic breakers* (HBs). The best properties for heuristic breaker gadgets are:

- **Use a small number of registers.** Such gadgets that preserve the values of registers allow us to chain multiple gadgets to carefully set the CPU and memory state to perform an operation like a call.

- **Used registers are loaded from memory or assigned constant values.** Long gadgets can have various side effects like loading and writing to

memory, etc.. When the registers used in such operations are set within the gadget, it is easier to prepare the gadgets so that the gadget does not cause a fatal fault.

- **Registers are loaded from memory.** Gadgets including the epilogue of functions frequently restore the values of various registers from the stack, allowing us to set multiple registers from the controlled stack.

- **Intended gadgets.** Long sequences of unintended gadgets tend to translate to unusual sequences of instructions. As a result, they are not as useful as intended gadgets. Our exploit, uses only a single small unintended gadget of two instructions, while another one only uses an unintended call instruction so it is call preceded.

Finally, we have to be flexible when a HBs cannot be included at a desired position in the gadget-chain. Consider for example a chain of seven gadgets. Ideally, we would insert a HB after the first five gadgets to break the sequence in two smaller ones, of five and two gadgets respectively. Since this is not always possible, due to the exploit's semantics, we may need to insert a HB sooner, for example *after* the first three gadgets.

4.4 Putting It All Together

Figure 4 provides a high-level graphical representation of our PoC exploit. We obtain control by exploiting the button object's VFT pointer, which grants us control of an indirect jump instruction. This instruction is actually part of a gadget (Appendix A, listing 1), as far as kBouncer is concerned, however it is not a gadget for ROPecker because it contains a conditional branch. The end goal is to invoke `VirtualProtect()` to mark the buffer we control and contains shellcode as executable. We can then transfer control to it, effectively bypassing DEP and performing a code-injection attack.

Our first task is to point the stack pointer (i.e., `ESP`) to the buffer we control, so we can perform ROP, a process commonly referred to as stack pivoting. After receiving control, `eax` points to our buffer, so we use an unintended gadget that exchanges the values of `eax` and `esp`, and terminates with a `ret`, to achieve this (Appendix A, listing 2).

Next, we want to prepare for calling `VirtualProtect()`. Before doing so, we need to interpose a HB gadget, so that the kBouncer check, triggered by entering the API function, will not detect our exploit. At this point, we know that the LBR contains two gadget addresses, the one for the stack pivoting gadget and the one before that, which is part of the program's legitimate control flow. We know that these two gadgets are not enough to cause detection, but there may be other entries in the LBR preceding these that could trigger kBouncer. Using a HB at this point ensures that the gadget-chain length in the LBR is reset. Moreover, using a HB at this point makes the payload generic, allowing us to use it with other vulnerabilities, as it will always break the gadget chain in the LBR, as long as T_G is less than its length. We use a HB gadget of 33 instructions that sets the `ESI` and `EDI` registers, which we use later on, and most importantly does not depend on any register being set up on entry (Appendix A, listing 3).

We perform the call to `VirtualProtect()` using a gadget that includes an indirect call (Appendix A, listing 4). This gadget only requires the `ESI` register to be prepared, which we set with the previous gadget. Also, it does not push any arguments to the stack, so the arguments to the call can be prepared in our buffer in advance. This is also the point where kBouncer kicks in and checks the LBR for an attack. By consulting Fig. 4, we notice that kBouncer cannot detect the attack at this point. When `VirtualProtect()` returns, control is transferred where it is expected to, that is, the instruction following the call. The next gadget executing is essentially the code following the indirect call (Appendix A, listing 5). The `ret` at the end of it transfers control to our shellcode, which is now executable. To ensure that we do not trigger any alarms in the future, we make sure that the first instruction in our shellcode is preceded by a fake, unused, call instruction.

Having managed to inject code into the process, we can now execute code without the risk of triggering kBouncer. Notice that this exploit will keep working even if T_G is raised to 31 and T_C reduced to 6.

5 Countermeasures

In this section, we discuss countermeasures, as well as fundamental boundaries in the use of gadget-chain length for preventing code-reuse attacks. While we mainly focus on kBouncer and ROPecker, we are confident that our analysis of their weaknesses and the proposed countermeasures will be of use to future works that plan to explore comparable methodologies.

5.1 Tweaking the T_G and T_C Parameters

An obvious improvement to both these techniques involves increasing T_G, i.e., the parameter that determines whether a sequence of instructions ending with an indirect branch is a gadget or not. Looking back at Fig. 4, it is clear that, in the case of kBouncer, increasing T_G to 33

instructions would neutralize our exploit. However, increasing the length of gadgets is not straightforward, as it has various side effects.

Increasing T_G will unavoidably lead to longer gadget chains that belong to legitimate, innocuous code. As we consider longer code sequence as potential gadgets, inevitably more application execution paths will be identified as gadgets, leading to observing longer gadget chains at run time. Consequently, to avoid false positives, T_C also needs to be increased to avoid misclassifying legitimate control flows as attacks. Unfortunately, raising T_C presents opportunities to attackers for using more gadgets. Both kBouncer and ROPecker assume that attackers cannot use longer gadgets due to the side effects that such gadgets have, something that both this paper and previous work [16] disproves. Defenders also face an asymmetry, as usual, because attackers need only find a handful of long gadgets to masquerade their payload. To maximize the effect of the parameters, what needs to be optimized is the fraction $\frac{T_G}{T_C}$. While this is probably an oversimplification, it provides a useful rule of thumb.

5.1.1 Per-Application Parameters

Acceptable settings for T_G and T_C vary depending on the application being examined [8]. The nature of the application itself, the compiler it was built with, and the shared libraries it uses, influence the generated binary code and what parameter values can be used to avoid FPs and concurrently detect attacks consistently.

We can exploit this observation to use different values based on the application. This would ensure that the strictest rules are applied every time. However, doing so is also not without difficulty because both the defense and the attack depend greatly on the application in question. For example, after analyzing an application, we can determine that a very strict set of T_C and T_G can be used without FPs. However, the application may contain gadgets much larger than T_G that can be directly chained together, so no gadget chains are identified. This scenario is obviously ideal for the attacker. Further research is required to establish a metric that quantifies the effect of selecting a particular set of parameters. Using per-application parameters can be also challenging in the presence of dynamically loaded (DL) libraries (i.e., libraries loaded at times other than program start up), as new, potentially unknown code is introduced in the application.

5.1.2 Per-Call Parameters

A novel idea to further customize the parameters is to use different gadget-chain thresholds (T_C) based on the part of the code executing. kBouncer that triggers check on certain API calls, would greatly benefit from this approach. Certain APIs may be normally called through limited executions paths that exhibit very particular characteristics. For instance, a Windows native API call (win32) is frequently called by higher-level frameworks. This approach has the benefit of both avoiding FPs, hence providing better stability, and improving security guarantees.

5.1.3 Cumulative Chain-Length Calculation

ROPecker also accumulates the lengths of smaller gadget-chain segments and uses a different parameter T_{CC} to detect attacks. While this heuristic is not effective with our exploit, it would be interesting to explore whether incorporating it in kBouncer, which checks for attacks less frequently and in a more controlled manner, would further raise the bar for attackers.

5.1.4 Obstacles

Recursive functions can cause significant problems with techniques based on counting gadget chains. Due to their nature, they can generate a large number of consecutive returns when they reach their end condition (e.g., when their computation has finished). If the returns within the recursive function lead to gadgets, then it is extremely hard to find any value of T_C that would not cause FPs, unless the recursion is very shallow (relative to the value of T_C). kBouncer seems to avoid such conditions because it only checks the LBR when an API call is made. In a sense, it performs checks at the boundary between application and kernel, and the intuition is that the checks are made "far" away from the algorithms in the core of applications. However, it is not an uncommon scenario that a recursive algorithm requires to allocate memory or write into a file. In such cases, lowering the maximum gadget length (T_G) is the only option for avoiding FPs. On the other hand, ROPecker performs checks far more frequently and whenever execution is transferred to new pages, so we expect that it is even more fragile in the presence of recursive algorithms.

5.2 Combining with CFI

CFI [1] enforces control-flow integrity and can prevent the exploit we describe in Sec. 4. In particular, recent CFI approaches like CCFIR [39] and binCFI [40] prevent the use of unintended gadgets, such as the two-instruction gadget we use for stack pivoting (Appendix A, listing 2). CCFIR, in particular, also disallows indirect calls to certain API calls like `VirtualProtect()`, so it prevents two gadgets used by our exploit. These CFI approaches also incur low performance overhead, making them a good candidate for

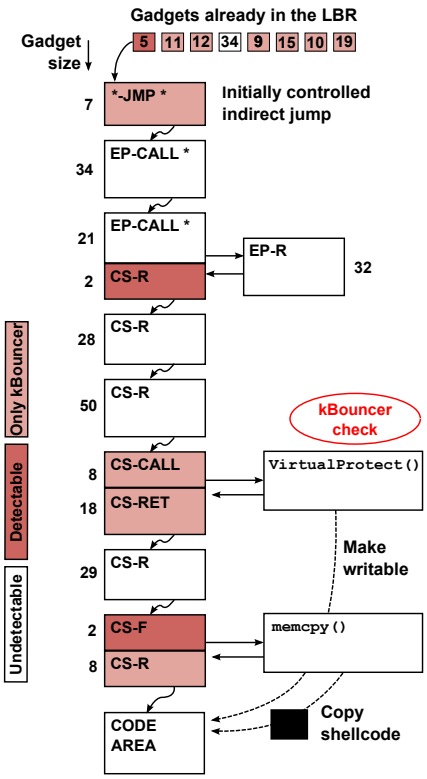


Figure 5: PoC exploit that bypasses kBouncer, ROPecker, and CCFIR. All the gadgets used are intended call-site (CS) or entry-point (EP) gadgets.

coupling with kBouncer. Unfortunately, recent work [16] has shown that they are still vulnerable to attack. *So is a combination of CFI and kBouncer still vulnerable?*

To answer the above question, we begin from the exploit used to bypass CFI in previous work [16, Sec. IV] and replace the smaller gadgets with longer HB gadgets that are also allowable by CFI. Similarly to prior work, we assume that CCFIR is in place, as it is stricter than binCFI. Because CFI does not allow transfers to new code, the goal of this payload is to mark existing code as writable and overwrite it with our shellcode. Before proceeding to describe the exploit, we summarize the additional restrictions imposed by CCFIR below.

Under CCFIR, return instructions can no longer transfer control to unintended gadgets, so only CS gadgets that were originally emitted by the compiler can be used when constructing a payload. Indirect call and jump instructions are also restricted and can only transfer control to function entry points, defining a new type of entry-point (EP) gadget. Indirect calls to sensitive API calls are prohibited, so any such calls need to be made using direct call instructions, contained within otherwise allowable gadgets. Finally, CCFIR introduces a new level of randomization through the use of springboard sections that proxy indirect control transfers. The location of these sections is randomized at load time and all indirect branches can only proceed through them.

To prepare the new payload, we need to replicate the steps described in Sec. 4, as well as a couple of additional steps required for bypassing CCFIR. Because of the randomized springboard sections, we need to reveal the sections that hold call and return stubs to the gadgets we plan to use. Fortunately, this can be achieved by exploiting the string object to leak code and meta-data from the DLLs of interest [16, Sec. IV.C]. `VirtualProtect()` and `memcpy()` are now called through gadgets that contain a direct call to these functions, so we do not need to explicitly locate them in the target process.

Figure 5 depicts a high-level overview of our second PoC exploit that overcomes the restrictions imposed by both kBouncer and CCFIR. We notice that because of CCFIR, we cannot use the same gadget to perform stack pivoting and, furthermore, we can only jump to an EP gadget. We resort to using a series of five gadgets to achieve the same goal (Appendix B, listings 6-10). Specifically, we first use three EP gadgets to corrupt the stack, so we can control a return instruction. This is similar to [16], however we use different, longer gadgets that are not detected by kBouncer. The fourth gadget loads `EBP` with our data and completes the switch to chaining through returns. We then use the fifth gadget, consisting of 28 instructions, to perform stack pivoting by copying `EBP` to `ESP` through the `leave` instruction.

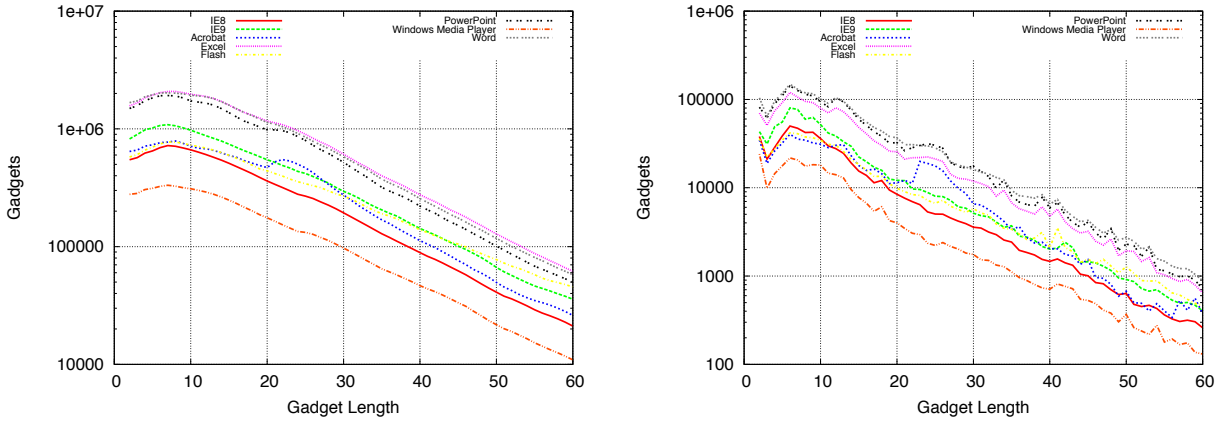
For calling `VirtualProtect()` and `memcpy()`, we reuse the same gadgets used against CFI (Appendix B, listings 12-13 and 15-16 respectively). The last gadget's `ret` transfers control to our shellcode that has been copied to the code section of the binary and has been preceded by a call to also foil future kBouncer checks. However, we replace the gadgets used to prepare the function-calling gadgets with the ones shown in listings 11 and 14 respectively, in Appendix B.

The exploit depicted in Fig. 5 demonstrates that even if we combine kBouncer and a loose CFI defense, it is still possible to devise attacks that can go undetected. Moreover, it shows that even if T_C and T_G are significantly tweaked, gadgets much larger than 21 instructions are available to partition an exploit to smaller, possibly undetectable, chains. In the particular exploit, setting T_G to 33 and T_C to 5 would still not have any effect.

6 Evaluation

6.1 Gadget Availability

The existence of long gadgets determines the potential to find HB gadgets that can be used to break long gadget chains. It is also an indicator on whether, we can poten-



(a) All gadgets.

(b) Call-preceded gadgets available to link using a `ret`.

Figure 6: Distribution of gadgets available to attackers under kBouncer based on their length, as recorded for multiple popular applications. We notice that there are numerous gadgets, even for gadgets of 60 instructions.

Application	**Workload**
Windows Media Player	Music playback for approximately 30 secs
Internet Explorer 9	Surf to google.com
Adobe Flash Player	Watch a YouTube video
Microsoft Word	Browse a Word document
Microsoft PowerPoint	Browse a PowerPoint presentation
Adobe Reader XI	Browse a PDF file

Table 1: Applications used in the evaluation.

tially use many different HB gadgets with varying functionality. We analyze the applications listed in Tab. 1, along with all their DLLs, to determine how many gadgets of different sizes they contain. In addition, we analyze Internet Explorer 8, which we used in our PoC. We follow the same methodology we used for collecting gadgets for the PoC exploit (Sec. 4.2). Specifically, we developed a gadget extraction tool in Python, using the popular *distorm* disassembler [15].

The process begins by disassembling from each byte in the code segment of each target binary, recursively following conditional branches, direct calls and jumps to locate instruction paths that end with a return or an indirect call or jump. That is, potential gadgets. As we disassemble, we count the number of instructions on each of the traversed paths, while we also keep track of the nodes we visit to avoid counting the same instructions more than once, due to loops. If we find more than one path starting from a particular byte and ending in an indirect branch, we keep the shortest path, and consider its length to be the length of the gadget at that byte. This is in accordance to how kBouncer identifies gadgets.

Figure 6 draws our results. We notice that even for relatively large gadget sizes, there are tens of thousands of gadgets. While we cannot make any assumptions on how usable they are, these results are an indication that there is a significant pool of gadgets to choose from. We attribute their large number to the fact that unintended gadgets are basically allowed by kBouncer.

6.2 Per-Application Parameters

In this section, we evaluate the feasibility of our per-application parameter scheme described in Sec. 5.1.1. To determine if, indeed, different applications can benefit from using tighter parameters, we run the six applications listed in Tab. 1 performing simple tasks, such as browsing. These applications were also used to perform a similar evaluation in kBouncer. We follow the same methodology to measure gadget-chain length.

We use a run-time monitoring tool based on Intel's Pin [22] to emulate the operation of LBR. We monitor every indirect branch instruction, including returns, jumps, and calls, and log the running thread ID, the address of the branch, and its target. To locate kBouncer gadgets, we borrowed the scripts used by kBouncer to disassemble the application images and their DLLs and, at the same time, locate the sensitive API calls where checks are injected by kBouncer [25, Appendix]. We combine the statically and dynamically collected information to match gadgets with control transfers observed at run time and calculate the length of gadget chains that would be checked by kBouncer.

Figure 7 shows the size of gadget chains (for $T_G = 20$), as they would be stored in LBR when entering a sensitive API call and for different applications. We observe that among the tested applications, only Adobe Reader

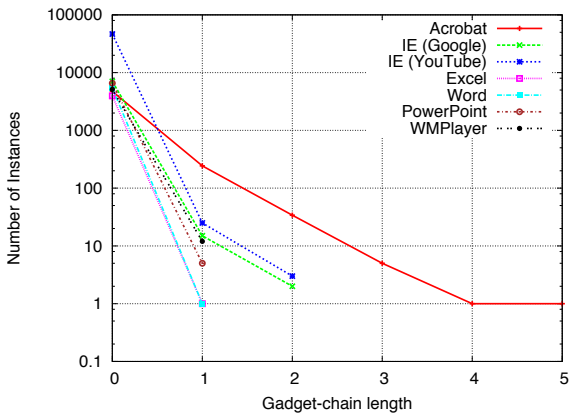


Figure 7: Length of chains for $T_G = 20$ for different applications, when entering a protected API function call. We run seven workloads using six applications, replicating the experiment in kBouncer [25, Sec. 3.2].

exhibits relatively long chains of five gadgets. This is in accordance with previously reported results [25]. All other tested applications include chains of up to two gadgets. In principle, this implies that we could use very strict values of T_C for all these applications. For example, using T_C = 3 with Internet Explorer would detect our long five-gadget sequence preceding the call to `VirtualProtect()` (Fig. 4).

This experiment shows that application-specific parameters can make it very hard for attackers to evade detection, at least with the methodology followed by our PoC exploit. However, we remain cautious, as run-time analysis can have limited coverage, even when divergent workloads are used, so using such strict values for T_C could cause FPs. In fact, recent works [12,29] report FPs with applications other than the ones originally tested in kBouncer [25], and when more restrictive parameters are employed. Nonetheless, establishing viable values for these parameters through dynamic and static analysis calls for additional research. The above serve as an indicator that per-application parameter fitting is necessary.

7 Related Work

Code reuse is the dominant form of exploitation since the wide adoption of stack canaries [9] and data execution prevention [2], which provide protection against stack smashing and code injections respectively. Return-to-libc attacks [37] is one of the simplest types of code reuse, involving the redirection of control to a `libc` function after setting up its arguments in the stack. Usually, this involves invoking functions like `system()` or `exec()` to launch another program (e.g., to spawn a shell). Short gadgets were also used in register springs [10] to load a register with the address of the attacker-controlled buffer located in the randomized stack or heap.

Return-Oriented Programming [31] generalizes the task of leveraging existing code to compromise a program. Short snippets of code, called gadgets, are chained together to introduce a new, not initially intended, control flow. ROP is particularly effective on instruction sets like CISC, where there are no instruction alignment requirements and the instruction number is high, because any sequence of executable bytes in memory can potentially become a gadget. Nevertheless, RISC architectures are also vulnerable to ROP [5].

Diversification approaches like Address-Space Layout Randomization (ASLR) [26] can be effective against ROP attacks and are already present in most OSs. ASLR randomizes the layout of a program when it is executed by loading the binary and its dynamic libraries on different base addresses each time. ASLR can be brute-forced [32], but the difficulty of doing so increases as more entropy becomes available, like in 64-bit systems. Recent attacks bypassing ASLR [30] rely on memory disclosure bugs that leak enough data from the targeted process to infer where the binary and/or its libraries are loaded at run time.

Finer-grained randomization approaches [17, 24, 38] have been proposed to further diversify programs and limit the effectiveness memory leaks. In-place randomization [24] relies on randomizing the sequence of instructions and replacing instructions with others of equivalent effect to further diversify the image of a running process. ILR [17] attempts to break the linearity of the address space, and binary stirring [38] randomizes a binary in the basic block level. However, recent research [34] has demonstrated that bugs that allow an attacker to read almost arbitrary memory locations can be used to bypass the above solutions as well.

CFI [1] enforces control-flow integrity preventing the malicious control flows that are part of ROP attacks, and it is not affected by memory leaks. CFI requires an accurate control-flow graph of the target program, which usually implies access to source code, but recent works [39, 40] have made steps towards addressing this limitation by applying a loose version of CFI on binaries. However, it has been recently shown [16] that these loose-CFI approaches are still vulnerable to attack in the presence of memory leaks. This work builds on the latter, borrowing the notion of call-site and entry-point gadgets, which are also the only kind of accessible gadgets under kBouncer and friends, and uses the same IE vulnerability as a starting point. However, the attack described in [16] is not effective against kBouncer. In this work we build an attack that is again effective. We also show that using the LBR and a set of heuristics is not sufficient to

prevent ROP attacks and reveal the inherent limitations of solutions based on gadget and chain length for detection. Finally, we propose various extensions that could alleviate the situation.

Compile-time solutions have been also proposed to alter the produced binary, so it is impervious to code reuse attacks. For example, producing a kernel that does not include any return instructions [21] cannot be exploited using ROP. However, variations of ROP that use indirect jumps instead of returns [3, 6] can be used to circumvent the above. G-Free [23] attempts to restrict control flow by using function cookies saved in a shadow stack at run time and inserts NOPs to destroy unintended gadgets. Because it is only loosely enforcing control flow, it is potentially vulnerable to the same attacks as CFI [16]. Additionally, compile-time approaches require that a binary and all of its libraries are recompiled.

Concurrently with our work, other efforts have also dealt with evaluating kBouncer and related approaches. Schuster et al. [29] take a slightly different approach and focus on finding gadgets that could be used to flush the LBR before performing any API call. The presence of such gadgets in the application nullifies any LBR-based defense, however, it leads to the same value being repeated in the LBR, which could potentially be used to detect such attacks. Moreover, the addition of CFI could restore the effectiveness of kBouncer.

On the other hand, Davi et al. [12] take an approach closer to ours. First, they show that there are enough small gadgets under loose CFI to perform any computation. Then, they introduce a long gadget of 23 instructions that does not perform any useful functionality and has limited side effects. They use this as a NOP gadget for breaking long gadget chains. Registers are not preserved, so additional gadgets need to be introduced to save any registers that need to be preserved. Larger NOP gadgets are not investigated, so unlike our approach their approach is more prone to detection when choosing stricter thresholds. Interestingly enough, both approaches test kBouncer, albeit with a different set of applications, and report false positives with the current, as well as with stricter thresholds.

8 Conclusion

In this paper we explored the feasibility of bypassing state-of-the-art ROP defenses based on monitoring processes (by means of Intel's new Last Branch Record) to detect control flows that resemble the execution of ROP chains [8, 25]. Essentially, these defenses check whether a sensitive API call was reached via a sequence of indirect branches to short, gadget-like, instruction sequences. Evading such detection is perceived as a hard task. First, all exploitation should be carried out using call-preceded gadgets, since otherwise the ROP chain will be easily detectable. Second, exploitation should find and then carefully insert long gadgets, in the middle of a series of shorter gadgets, in order to fly under the defense's ROP radar. The long gadgets should be long enough to make the ROP chain look like a legitimate control flow of the running process. Finding such long gadgets and gluing them in the actual ROP chain is not trivial, since it is possible that these long series of instructions interfere with the state of the exploit (e.g., modify a valuable register). Nevertheless, in this paper, we successfully constructed two real exploits, which utilizes the long gadgets to evade detection. With this work we stress that the selection of critical parameters, such as the length of a series of instructions that should be considered a gadget, as well as the gadget-chain length is not trivial. Until we solve these problems, the defenses are prone to false negatives and false positives. Finally, we discuss various countermeasures and provide evidence, through an experimental evaluation, that defining parameters on a per-application basis, can alleviate these concerns.

Acknowledgment

We want to express our thanks to the anonymous reviewers for their valuable comments. In particular, we want to thank our shepherd, Kevin Fu, who helped us give this paper its final form. This work was supported by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, or the Air Force. This work was also supported in part by the ERC StG project Rosetta, the FP7-PEOPLE-2010-IOF project XHUNTER, No. 273765, the Prevention of and Fight against Crime Programme of the European Commission – Directorate-General Home Affairs (project GCC), and EU FP7 SysSec, funded by the European Commission under Grant Agreement No. 257007.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proc. of the 12th ACM CCS* (2005).

[2] ANDERSEN, S., AND ABELLA, V. Changes to functionality in Microsoft Windows XP Service Pack 2, part 3: Memory protection technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004. `http://technet.microsoft.com/en-us/library/bb457155.aspx`.

[3] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proc. of the 6th ACM ASIACCS* (2011).

[4] BOSMAN, E., AND BOS, H. We got signal. a return to portable exploits. In *Proc. of the 35th IEEE S&P* (2014).

[5] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proc. of the 15th ACM CCS* (2008).

[6] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proc. of the 17th ACM CCS* (2010).

[7] CHEN, P., XIAO, H., SHEN, X., YIN, X., MAO, B., AND XIE, L. DROP: Detecting return-oriented programming malicious code. In *Proc. of the 5th ICISS* (2009).

[8] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND DENG, R. H. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proc. of the 21st NDSS* (2014).

[9] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., ET AL. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium* (1998).

[10] DARK SPYRIT. Win32 buffer overflows (location, exploitation, and prevention). *Phrack magazine 9*, 55 (1999).

[11] DARKREADING. Heap spraying: Attackers' latest weapon of choice. `http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428`, November 2009.

[12] DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proc. of the 23rd USENIX Security Symposium* (August 2014).

[13] DAVI, L., SADEGHI, A.-R., AND WINANDY, M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proc. of the 6th ACM ASIACCS* (2011).

[14] DEMOTT, J. Bypassing emet 4.1. `http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf`, February 2014.

[15] DISTORM. Powerful disassembler library for x86/AMD64. `https://code.google.com/p/distorm/`.

[16] GÖKTAŞ, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proc. of the 35th IEEE S&P* (May 2014).

[17] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd my gadgets go? In *Proc. of the 33rd IEEE S&P* (2012).

[18] HITMANPRO. Real-time exploit mitigation and intrusion detection. `http://dl.surfright.nl/Alert-3/HitmanPro-Alert-3-Datasheet.pdf`, February 2014.

[19] HUNT, G., AND BRUBACHER, D. Detours: Binary interception of Win32 functions. In *Proc. of the 3rd Conference on USENIX Windows NT Symposium* (1999).

[20] INTEL. Intel 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide, part 2. `http://www.intel.com`.

[21] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. Defeating return-oriented rootkits with return-less kernels. In *Proc. of the 5th EuroSys* (2010).

[22] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the 26th PLDI* (2005).

[23] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th ACSAC* (2010).

[24] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. of the 33rd IEEE S&P* (2012).

[25] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proc. of the 22nd USENIX Security Symposium* (2013).

[26] PAX TEAM. Address Space Layout Randomization, 2003. `http://pax.grsecurity.net/docs/aslr.txt`.

[27] PELLETIER, A. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). VUPEN Vulnerability Research Team (VRT) Blog, July 2012. `http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php`.

[28] PORTNOY, A. Bypassing all of the things. EXODUS INTELLIGENCE. `https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf`.

[29] SCHUSTER, F., TENDYCK, T., PEWNY, J., MAASS, A., STEEGMANNS, M., CONTAG, M., AND HOLZ, T. Evaluating the effectiveness of current anti-ROP defenses. In *Proc. of the International Conference on RAID* (September 2014).

[30] SERNA, F. J. CVE-2012-0769, the case of the perfect info leak. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

[31] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM CCS* (October 2007).

[32] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM CCS* (2004).

[33] SKOWYRA, R., CASTEEL, K., OKHRAVI, H., ZELDOVICH, N., AND STREILEIN, W. Systematic analysis of defenses against return-oriented programming. In *Proc. of the 16th RAID* (2013).

[34] SNOW, K. Z., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., MONROSE, F., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of the 34th IEEE S&P* (May 2013).

[35] SOTIROV, A. Heap feng shui in javascript. *Black Hat Europe* (2007).

[36] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *Proc. of the 2nd EuroSec* (2009).

[37] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *Proc. of the 14th RAID* (2011).

[38] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of the 2012 ACM CCS* (2012).

[39] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proc. of the 34th IEEE S&P* (2013).

[40] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *22nd USENIX Security Symposium* (2013).

A PoC Exploit Gadgets

```
; library: mshtml.dll
; offset:  0x001BC907
; type:    intended, *-JMP *
1 mov   eax, [ecx+1Ch]
2 test  al,  al
3 js    loc1
```

```
loc1:
4 mov   ecx, dword ptr [ecx+24h]
5 mov   eax, dword ptr [ecx]
6 mov   edx, dword ptr [eax+24h]
7 jmp   edx                 ; edx points to 1st gadget
```

Listing 1: The exploitation of the vulnerability in Internet Explorer 8 gives us control of an indirect jump. The target address of the indirect jump is loaded from the sprayed buffer, by dereferencing values in the overwritten button object. After this code sequence, `ecx` contains a pointer to the overwritten button object, `eax` contains a pointer to our sprayed buffer, and `edx` contains the address of the first gadget.

```
; library: shell32.dll
; offset:  0x00146FB2
; type:    unintended, *-RET
1 xchg  esp, eax
2 retn
```

Listing 2: This is the first executed gadget after the control of the indirect jmp instruction. The gadget performs the stack pivoting operation. Essentially, the values in the `eax` and the `esp` registers are swapped. On entry, `eax` points to the sprayed buffer, which contains the rest of the ROP chain.

```
; library: shell32.dll
; offset:  0x0007AACD
; sort:    intended instructions, unintended CS-RET
 1 mov     esi, 738AD720h
 2 mov     edi, 73BCC3C0h
 3 movsd
   ...
 7 mov     esi, 738AD710h
 8 mov     edi, 73BCC3D4h
 9 movsd
   ...
13 mov     esi, 738AD730h
14 mov     edi, 73BCC3E8h
15 movsd
   ...
19 mov     esi, 738AD700h
20 mov     edi, 73BCC3FCh
21 movsd
   ...
25 mov     esi, 7387A2CCh
26 mov     edi, 73BCC410h
27 movsd
   ...
31 pop     edi
32 pop     esi
33 retn
```

Listing 3: This is a heuristic-breaker gadget, i.e., an undetectable long gadget. First, it is used to reset the chain of detectable gadgets in the ROP chain. Second, it will prepare the `esi` register, which is required by the next gadget that will call `VirtualProtect()`. Upon entry, it does not require any registers to be already set up, but it alters two registers: `esi` and `edi`, loading them with values from our buffer.

```
; library: shell32.dll
; offset:  0x0039C0E5
; type:    intended, CS-CALL *
1 lea     ecx, [esi+28h]
2 mov     edi, eax
3 mov     eax, [ecx]
4 call    dword ptr [eax+44h]
```

Listing 4: An indirect function call that we use to call `VirtualProtect()` and change the memory permissions of the region occupied by the injected shellcode, which also resides in the sprayed buffer. The gadget does not push values, so the arguments for the called function can be prepared in advance in the ROP chain, and it also saves `EAX` in `EDI` before calling.

```
; library: shell32.dll
; offset:  0x0039C0EF
; type:    intended, CS-RET
1 mov     eax, edi
2 pop     edi
3 pop     esi
4 pop     ebp
5 retn    0Ch
```

Listing 5: The instructions following the indirect call in listing 4 also constitute a gadget. This gadget restores `EAX` from `EDI`, thus restoring to the value it had before entering the previous gadget, and returns using the next value in our ROP chain transferring control to our shellcode.

B CFI-resistant PoC Exploit Gadgets

```
; library: ieframe.dll
; offset:  0x00216C0E
; type:    EP
 1 mov     edi, edi
 2 push    ebp
 3 mov     ebp, esp
 4 sub     esp, 2C8h
 5 mov     eax, ___security_cookie
 6 xor     eax, ebp
 7 mov     [ebp-4], eax
 8 mov     eax, [ebp+0Ch]
 9 push    ebx
10 push    esi
11 push    edi
12 mov     edi, [ebp+8]
13 mov     [ebp-290h], eax
14 xor     eax, eax
15 push    3
16 mov     [ebp-280h], eax
17 mov     [ebp-284h], eax
18 mov     [ebp-288h], eax
19 mov     [ebp-2A0h], eax
20 mov     [ebp-2A4h], eax
21 pop     eax
22 mov     esi, ecx
23 mov     [ebp-2B8h], ax
24 mov     eax, [esi+24h]
25 lea     edx, [ebp-2C8h]
26 push    edx
27 mov     [ebp-2B0h], eax
28 mov     eax, [esi+1Ch]
29 mov     ecx, [eax]
30 lea     edx, [ebp-2B8h]
31 push    edx
32 push    eax
33 mov     [ebp-294h], edi
34 call    dword ptr [ecx+1Ch]
```

Listing 6: By pushing a pointer to the sprayed buffer as an argument (see Line 32), this gadget prepares the gadget (see Listing 7) that will call the stack smasher (see Listing 8).

```
; library: mshtml.dll
; offset:  0x004A959F
; type:    EP
 1 mov     edi, edi
 2 push    ebp
 3 mov     ebp, esp
 4 push    dword ptr [ebp+30h]
 5 mov     eax, [ebp+8]
 6 push    dword ptr [ebp+2Ch]
 7 mov     ecx, [eax+4]
 8 push    dword ptr [ebp+28h]
 9 mov     ecx, [ecx]
10 push    dword ptr [ebp+24h]
11 push    dword ptr [ebp+20h]
12 push    dword ptr [ebp+1Ch]
13 push    dword ptr [ebp+18h]
14 push    dword ptr [ebp+14h]
15 push    dword ptr [eax+0Ch]
16 push    dword ptr [eax+8]
17 push    dword ptr [ebp+10h]
18 push    dword ptr [ebp+0Ch]
19 push    eax
20 push    dword ptr [ecx]
21 call    dword ptr [ecx+10h]
```

Listing 7: This gadget will push the address of the call site gadget (see Line 15) we want to be executed later in the chain (see Listing 10). Once we get to this desired call site gadget, the switch from Entry Point to Call Site gadgets is complete.

```
; library: ieframe.dll
; offset:  0x000A98B5
; type:    EP
 1 mov     edi, edi
 2 push    ebp
 3 mov     ebp, esp
 4 mov     eax, [ebp+8]
 5 mov     ecx, [eax+140h]
 6 push    ebx
 7 mov     ebx, [ebp+14h]
 8 push    esi
 9 mov     esi, [ebp+0Ch]
10 mov     [esi], ecx
11 lea     ecx, [eax+144h]
12 mov     edx, [ecx]
13 push    edi
14 mov     edi, [ebp+10h]
15 mov     [edi], edx
16 lea     edx, [eax+148h]
17 mov     edi, [edx]
18 mov     [ebx], edi
19 xor     edi, edi
20 mov     [eax+140h], edi
21 mov     [ecx], edi
22 mov     [edx], edi
23 mov     eax, [esi]
24 neg     eax
25 sbb     eax, eax
26 pop     edi
27 and     eax, 7FFFBFFBh
28 pop     esi
29 add     eax, 80004005h
30 pop     ebx
31 pop     ebp
32 retn    10h
```

Listing 8: This is a long gadget that moves data and does not harm the status of our ROP chain. Also, it will break the calling assumptions of the caller gadget (see Listing 7).

```
; library: mshtml.dll
; offset:  0x004A95D6
; type:    EP
 1 pop     ebp
 2 retn    2Ch
```

Listing 9: The instructions following the indirect call in listing 7 also constitute a gadget. The return instruction in this gadget will use the address of the call site gadget that was pushed before (see Listing 7). Also, in this gadget `ebp` is prepared with a pointer to our sprayed buffer. The value in this register will be moved to the `esp` register in the stack pivoting gadget (see Listing 10).

```
; library: mshtml.dll
; offset:  0x00305202
; type:    CS
 1 mov     ecx, [ebp+30h]
 2 mov     edx, [ebp+40h]
 3 mov     [ecx], eax
 4 mov     eax, [ebp+34h]
 5 mov     ecx, [ebp+3Ch]
 6 shr     esi, 6
 7 and     esi, 1
 8 and     dword ptr [ebp+8], 0
 9 mov     [eax], esi
10 mov     eax, [ebp-0Ch]
11 mov     [ecx], eax
12 mov     eax, [ebp-4]
13 mov     ecx, [eax]
14 mov     ecx, [ecx+88h]
15 mov     [edx], ecx
16 mov     ecx, [eax]
17 mov     ecx, [ecx+30h]
18 mov     edx, [ebp+44h]
19 mov     [edx], ecx
20 mov     ecx, [ebp+38h]
21 mov     [ecx], eax
22 jmp     next_ins
23 mov     eax, [ebp+8]
24 pop     edi
25 pop     esi
26 pop     ebx
27 leave
28 retn    40h
```

Listing 10: This is a stack pivoting gadget. This gadget will load `esp` with a pointer to our sprayed buffer at Line 27.

```
; library: mshtml.dll
; offset:  0x0021FDF4
; type:    CS
 1 mov     eax, [ebp+0Ch]
 2 mov     [ebx], eax
 3 push    7
 4 pop     ecx
 5 lea     esi, [eax+228h]
 6 rep movsd
 7 mov     ecx, [eax+244h]
 8 mov     [ebx+20h], ecx
 9 mov     ecx, [eax+260h]
10 mov     [ebx+24h], ecx
11 mov     ecx, [eax+264h]
12 mov     [ebx+28h], ecx
13 mov     ecx, [eax+268h]
14 mov     [ebx+2Ch], ecx
15 mov     ecx, [eax+26Ch]
16 mov     [ebx+30h], ecx
17 mov     ecx, [eax+270h]
18 mov     [ebx+34h], ecx
19 mov     ecx, [eax+274h]
20 mov     [ebx+38h], ecx
21 mov     ecx, [eax+278h]
22 mov     [ebx+3Ch], ecx
23 mov     ecx, [eax+27Ch]
24 mov     [ebx+40h], ecx
25 mov     ecx, [eax+280h]
```

```
26 mov     [ebx+44h], ecx
27 mov     ecx, [eax+284h]
28 mov     [ebx+48h], ecx
29 mov     ecx, [eax+288h]
30 mov     [ebx+4Ch], ecx
31 mov     ecx, [eax+28Ch]
32 mov     [ebx+50h], ecx
33 mov     ecx, [eax+290h]
34 mov     [ebx+54h], ecx
35 mov     ecx, [eax+2CCh]
36 mov     [ebx+58h], ecx
37 mov     ecx, [eax+2D0h]
38 mov     [ebx+5Ch], ecx
39 mov     ecx, [eax+2D4h]
40 mov     [ebx+60h], ecx
41 mov     ecx, [eax+2D8h]
42 mov     [ebx+64h], ecx
43 mov     eax, [eax+2DCh]
44 pop     edi
45 mov     [ebx+68h], eax
46 pop     esi
47 mov     eax, ebx
48 pop     ebx
49 pop     ebp
50 retn    8
```

Listing 11: This is a long Heuristic Breaker gadget. It will also prepare the `ebp` and `ebx` registers which are required by the VirtualProtect calling function (see Listing 12).

```
; library: ieframe.dll
; offset:  0x0006FBAE
; type:    CS
 1 and     dword ptr [ebp-0Ch], 0
 2 lea     eax, [ebp-0Ch]
 3 push    eax             ; old protection
 4 push    40h             ; new protection
 5 push    ebx             ; size
 6 mov     ebx, [ebp-8]
 7 push    ebx             ; address
 8 call    ds:VirtualProtect
```

Listing 12: Gadget that makes program code writeable, whereto inject a shellcode later. The part after the `call` instruction is considered as a separate gadget as it is the target of an indirect branch (i.e., return instruction of the `VirtualProtect` function).

```
; library: ieframe.dll
; offset:  0x0006FBC3
; type:    CS
 1 test    eax, eax
 2 jz      loc_766E9531 ; if eax==0: handle error
 3 mov     eax, [ebp+8]
 4 and     dword ptr [edi+4], 0
 5 mov     [edi+8], eax
 6 mov     [edi+10h], esi
 7 mov     [edi+0Ch], ebx
 8 mov     eax, [ebp-0Ch]
 9 mov     [edi+14h], eax
10 mov     eax, dword_768E2CCC
11 mov     [edi], eax
12 mov     dword_768E2CCC, edi
13 xor     eax, eax
14 pop     edi
15 pop     ebx
16 pop     esi
17 leave               ; == mov esp, ebp and pop ebp
18 retn    14h
```

Listing 13: Gadget that makes program code writeable, whereto inject a shellcode later. The part after the `call` instruction is considered as a separate gadget as it is the target of an indirect branch (i.e., return instruction of the `VirtualProtect` function)

```
; library: mshtml.dll
; offset:  0x000DA72B
; type:    CS
 1 mov     eax, [ebp+10h]
 2 mov     [ebx+108h], esi
 3 mov     esi, [ebp+8]
 4 lea     edi, [ebx+110h]
 5 movsd
   ...
 9 mov     esi, [ebp+0Ch]
10 push    0Dh
11 pop     ecx
12 lea     edi, [ebx+120h]
13 rep movsd
14 mov     [ebx+154h], eax
15 mov     eax, [ebp+1Ch] !!
16 mov     ecx, [eax]
17 push    0Dh
18 mov     [ebx+0C0h], ecx
19 pop     ecx
20 lea     esi, [eax+18h]
21 lea     edi, [ebx+0C4h]
22 rep movsd
23 mov     eax, [eax+4Ch]
24 pop     edi
25 pop     esi
26 mov     [ebx+0F8h], eax
27 pop     ebx
28 pop     ebp
29 retn    18h
```

Listing 14: Another Heuristic Breaker gadget and at the same time it will prepare `eax` for the memcpy calling gadget.

```
; library: ieframe.dll
; offset:  0x001ADCC2
; type:    CS
1 push    eax                ; destination
2 call    memcpy
```

Listing 15: Gadget for calling of `memcpy` for copying the shellcode to existing program code.

```
; library: ieframe.dll
; offset:  0x001ADCC8
; type:    CS
1 add     esp, 0Ch
2 xor     eax, eax
3 jmp     short loc_7672DCE7
4 pop     ebx
5 pop     edi
6 pop     esi
7 pop     ebp
8 retn    8
```

Listing 16: The call site part of the memcpy calling gadget.

Oxymoron
Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing

Michael Backes
Saarland University, Germany
Max-Planck-Institute for Software Systems, Germany
backes@mpi-sws.org

Stefan Nürnberger
Saarland University, Germany
nuernberger@cs.uni-saarland.de

Abstract

The latest effective defense against code reuse attacks is fine-grained, per-process memory randomization. However, such process randomization prevents code sharing since there is no longer any identical code to share between processes. Without shared libraries, however, tremendous memory savings are forfeit. This drawback may hinder the adoption of fine-grained memory randomization.

We present Oxymoron, a secure fine-grained memory randomization technique on a per-process level that does not interfere with code sharing. Executables and libraries built with Oxymoron feature *'memory-layout-agnostic code'*, which runs on a commodity Linux. Our theoretical and practical evaluations show that Oxymoron is the first solution to be secure against just-in-time code reuse attacks and demonstrate that fine-grained memory randomization is feasible without forfeiting the enormous memory savings of shared libraries.

1 Introduction

Code reuse attacks manage to re-direct control flow through a program with the intent of imposing malicious behavior on an otherwise benign program. Despite being introduced more than 20 years ago, code reuse is still one of the three most prevalent attack vectors [1, 28], e.g., through vulnerable PDF viewers, browsers, or operating system services. Several code reuse mitigations have been proposed. They either detect the redirection of control flow [7, 12], or randomize a process's address space. Randomizations jumble the whole address space, with the intent of preventing code reuse attacks by making it impossible to predict where specific code resides.

Especially *Address Space Layout Randomization* (ASLR [23, 22]) has become widespread, but meanwhile has been shown to be ineffective [24, 25]. A promising avenue is the use of even finer randomization techniques that randomize at the granularity of functions, basic blocks or even instructions [18, 10, 16, 19, 17].

To be effective, fine-grained memory randomization must prevent an attacker from using information about the memory layout of one process to infer the layout of another process. This is a particular threat in the light of shared code originating from shared libraries. Hence, most recent fine-grained memory randomization solutions also randomize shared libraries for every single process [13, 21, 29]. As a result, there is no identical code in any two processes, which makes sharing impossible. A dysfunctional code sharing, however, increases the memory footprint of the entire system, likely on the order of Gigabytes, as we elaborate in Section 2.

To summarize: fine-grained randomization solutions presented so far come at the expense of tremendous memory overhead, which renders them impractical.

Oxymoron /ˌɒk.sɪ'mɔː.rɒn/ (noun)
Greek. A figure of speech that combines contradictory terms.

We present Oxymoron, which combines two seemingly contradictory methods: a secure fine-grained memory randomization with the ability to share the entire code among other processes. At the heart of Oxymoron is a new x86 calling convention we propose: *Position-and-Layout-Agnostic CodE (PALACE)*. This code uses no instructions that reference other code or data directly, but instead the instructions use a layer of indirection referred to by an index. This index uniquely identifies a target and hence remains identical when targets are randomized in memory. Consequently, the memory in which those instructions are stored does not change, thereby making it available to be shared with other processes.

Oxymoron cuts program code into the smallest sharable piece: a memory page. We randomize those pages and share them individually among processes. Each shared page appears at a different, random address in each process. We use the x86 processor's segmentation feature to disable access to the unique indices, which we organized in a translation table. This unique property of Oxymoron makes our solution more secure than fine-grained memory randomization solutions published so far.

To demonstrate the effectiveness and efficiency of Oxymoron, we implemented and evaluated a static binary rewriter for the Intel x86 architecture that emits PALACE executables and libraries with a very low run-time overhead of only 2.7%. By re-enabling code sharing, Oxymoron is the first memory randomization technique that reduces the total system memory overhead back to levels it was before fine-grained memory randomization, while simultaneously being the first solution that is secure against the just-in-time code reuse attacks recently proposed by Snow et al [26].

2 Problem Description

Before we describe our idea, we want to explain in more detail why any traditional fine-grained memory randomization necessarily makes sharing libraries impossible. The goal of fine-grained randomization is for every process to feature a memory layout that is as varied as possible from any other process. If we treat program code, which usually is en bloc, as a puzzle and shuffle the puzzle pieces throughout the entire address space, their combinatorial possibilities provide a high entropy. It is only possible to share those puzzle pieces individually as a memory page with other processes if the content of each piece is identical in each process. With traditional code, the content of those piece necessarily changes when their order in memory is rearranged, as we explain in the following:

Code references other code or other data using either absolute memory addresses, e.g., `call 0x804bd32`, or relative addresses, e.g., `call +42`. For absolute addresses it is obvious that different randomizations necessarily lead to different code and data addresses. As a result, the encoding of instructions that hold such addresses changes as well, thereby forfeiting the sharing with other processes. Relative addresses, in turn, make code independent of its load address in memory. However, in case of using code pieces that are randomized, the relative distances change as well. Here, for the same reason, those pieces cannot be shared across processes as they feature different relative addresses. Consequently, fine-grained memory randomization impedes common code sharing, which is a fundamental concept of *all* modern OSes.

Severity. Modern operating systems use code sharing automatically, and it is in effect because the running programs use the same libraries (C library, threading library etc.), i.e. their address spaces have identical code loaded.

To verify this claim, we conducted a simple experiment that shows the impact of code sharing and lack thereof. We used an unmodified Ubuntu 13.10 x86 operating system on a machine with 4 GB of RAM and evaluated how much RAM is saved due to code sharing. After booting to an idle desktop, the 234 running processes consumed a total 679 MB. Our analysis of memory page mappings in each process obtained from `/proc/<PID>/maps` revealed that most of the processes used the same set of shared libraries. As expected, most frequently the standard C-library `libc.2.17.so` was shared between all of the 234 processes. All mapped portions of `libc` sum up to 207,028 KB while only 885 KB of real memory are consumed. This is a savings of 206 MB for `libc` alone.

Figure 1 illustrates the top ten savings by library. In total, sharing instead of duplicating saved 1,388 MB of RAM on the idle Ubuntu desktop. When additionally starting the Firefox browser, the memory consumption was increased from 679 MB to 817 MB. The total amount of savings by sharing summed up to 1,435 MB of RAM, which is an additional savings of 47 MB.

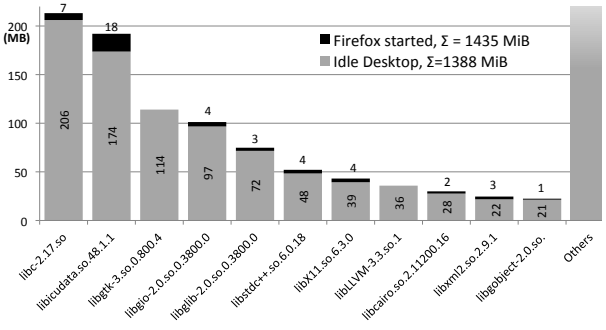


Figure 1: *Savings due to sharing of libraries. Idle desktop saves 1388 MB, with Firefox 47 MB more is saved.*

2.1 Threat Model

We assume a Linux operating system that runs a user mode process, which contains a memory corruption vulnerability. The attacker's goal is to exploit this vulnerability in order to divert the control flow and execute arbitrary code on her behalf. To this end, the attacker knows the process' binary executable and can precompute potential gadget chains in advance. The attacker can control the input of all communication channels to the process, especially including file content, network traffic, and user input. However, we assume that the attacker

has not gained prior access to the operating system's kernel and that the program's binary is not modified. Apart from that, the computational power of the attacker is not limited.

Moreover, for JIT-ROP attacks [26] to work, we assume that the process has at least one memory disclosure vulnerability, which makes the process read from an arbitrary memory location chosen by the attacker and report the value at that location. This vulnerability can be exploited any number of times during the runtime of the process. Note that the process itself performs the read attempt: both address space and permissions are implied to belong to the process.

3 High-Level Design of Oxymoron

To benefit from the best of both worlds – fine-grained memory randomization and code sharing – the challenge is to create a form of code that does not incorporate absolute or relative addresses, as we have already shown that both addressing schemes by definition suffer from being dependent on their randomization. An additional layer of indirection that translates unique labels to current randomized addresses allows the byte representation of code to remain the same, which enables code page sharing. However, this approach is difficult to realize as it is accompanied by four key challenges:

1. keeping the size of the translation table small in order not to increase the memory that we saved,
2. developing an efficient layer of indirection so that it is practical,
3. making the translation inaccessible by adversaries,
4. making the solution run on a commodity, unmodified Linux OS.

Overall Procedure. Oxymoron prevents code reuse attacks by shuffling every instruction of a program to a completely different position in memory so that no instruction stays at a known address, thereby making it infeasible for an adversary to guess or brute-force addresses. We use a three-step procedure (cf. Figure 2):

A) **Code Transformation**: The executable E is transformed to Position-and-Layout-Agnostic CodE (PALACE). The result is a PALACE-code executable P_E. The same applies to shared libraries, which can be treated like executables.

B) **Splitting**: The P_E code is then split into the smallest possible piece that can be shared among processes: a memory page. The code of P_E now consists of code pieces $P_E = p_1|p_2|\ldots|p_n$.

C) **Randomization**: At program load time, the pieces $p_1|p_2|\ldots|p_n$ are shuffled by the ASLR part of the operating system loader. In memory, their order is completely random and the pieces may have empty gaps of arbitrary size between them.

The first two steps only have to be done once, while the third step is performed at load-time of the executable P_E.

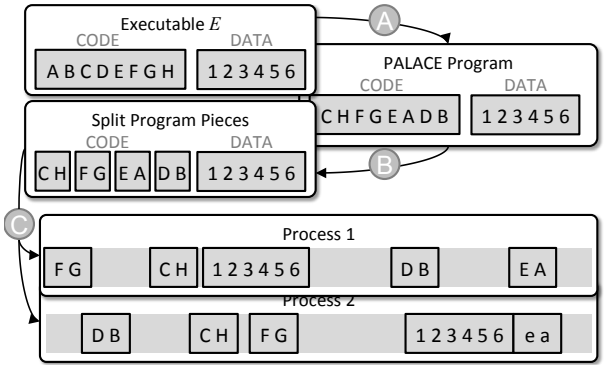


Figure 2: *The program is transformed and split once (A and B), then randomized at every process start-up (C).*

3.1 Code Transformation

To enable layout-agnostic code, all references to code and data are replaced with a unique label. Such a unique label is an assigned index into a translation table. This *Randomization-agnostic Translation Table* (RaTTle) in turn refers to the actual target (see Figure 3). This is the key to code sharing among processes, since the byte representation of the PALACE code does not change in the next step, when it is split and individual pieces are shuffled in memory.

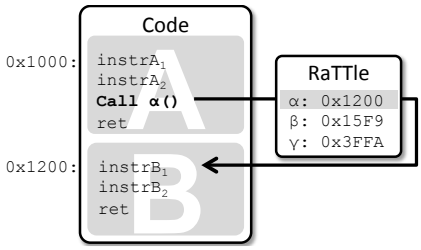


Figure 3: *Control-flow is redirected through the RaTTle rather than jumping to addresses directly.*

3.2 Splitting

Splitting ensures that the resulting pieces can be mapped into different processes at different addresses. As PALACE code references every target through a unique label in the RaTTle, it can be split without the need for traditional relocation, which rewrites addresses that hence have changed.

The PALACE code is split into page-sized pieces. If those pieces are later shuffled, it must be assured that the original semantics of the program are kept intact. This is essential when control flows from the end of one piece to the piece that was adjacent to it in the original program code. Thus, we need to insert explicit control flows between consecutive code pieces that might be moved away in a later stage of randomization. These explicit links only need to be inserted as the last instruction of a piece to ensure that control indeed flows to the intended successor (see Figure 4). After the links have been inserted, the code pieces can be randomized in memory without violating the original program semantics.

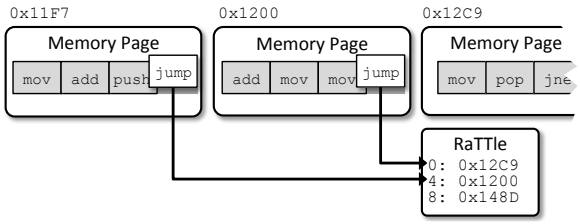


Figure 4: *Filling a page with instructions and linking them with explicit control flow transfers.*

3.3 Randomization

Modern OS loaders for shared libraries already support Address Space Layout Randomization (ASLR), i.e. they load the code, data, and stack segments at random base addresses. We leverage this fact by putting every memory page in its own loadable segment of the executable file or of the shared library. As the page-sized code pieces are already transformed to PALACE code, no relocation of addresses is needed. An ASLR-enabled commodity loader can blindly load all pieces at random addresses. Consequently, each process can have its own permutation of the randomization. Only the RaTTle needs to be kept up to date with a per-process randomization (see *"Populating the RaTTle"*).

3.4 Addressing the RaTTle

At first glance, it might seem we have only shifted the problem of addressing functions in code to securely addressing the RaTTle. However, our approach enables secure access to the RaTTle without access for adversaries. We first explain why we chose the more involved realization of the RaTTle and not existing approaches, such as a fixed address, a fixed register or the Global Offset Table (GOT). As already alluded to by Shacham et al. [25], the following techniques have drawbacks:

Fixed. Storing the RaTTle at a fixed address in memory allows for its address to be hard-coded in the instructions themselves. Unfortunately, a hard-coded address restricts the table to a fixed position. This fact can be exploited by an attacker.

GOT. Accessing the GOT is realized by using relative addresses, which forfeits sharing as discussed earlier. Moreover, several attacks are known that dereference the GOT [5].

Register. A dynamic address that is randomly chosen for every process could be stored in a dedicated machine register. However, this register would need to be sacrificed and every original use of that register must then be simulated with other registers or the stack. Moreover, a leakage vulnerability could reveal the address of the RaTTle.

Our Approach. Our RaTTle does not suffer from the aforementioned drawbacks. We use the x86 feature of memory *segmentation* to address and at the same time hide the RaTTle from adversaries. X86's segmentation is disused today because it has been superseded by memory paging. Memory paging, also called virtual memory, allows a fine-grained mapping of memory on a per-process basis and is much more versatile than segmentation. However, segmentation is still available in modern processors and in combination with paging allows for the security we need for the RaTTle. Additionally, as segmentation is a hardware feature and we can use it to implement the translation table, it is very efficient. Segmentation allows the memory to be divided in user-defined segments that may overlap. Segmentation is realized in the processor by adding a user-defined offset to all addresses the code handles (see Figure 5).

Segmentation allows different so-called *segment descriptors* to be created, each with their own *base address* and *limit*, i.e. the start and length of that segment. The list of these segment descriptors is kept in the *Global Descriptor Table* (GDT, see Figure 5). *Segment selectors* must then point to exactly one segment entry in that GDT. Segment selection is done using dedicated segment selector registers such as `CS` (Code Segment), `DS` (Data Segment),

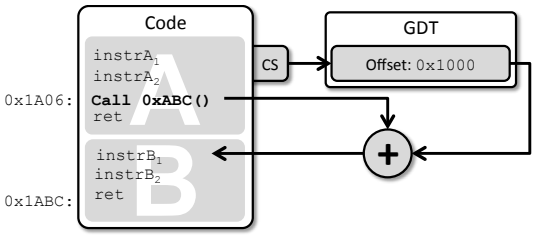


Figure 5: *Code using segments as offsets for addresses.*

SS (Stack Segment) and three general purpose segment selectors ES, FS and GS.

Position-and-Layout-Agnostic CodE (PALACE). The trick we use is the fact that segments can be selectively overridden on a per-instruction basis. In this way, a single instruction may use an addressing that is relative to the RaTTle, thereby indexing the RaTTle to change control flow or to access data. For example, `call *%fs:0x4` dereferences the double-word stored at `%fs:4` and calls the function stored at that double-word. If we let the segment selector FS point to the randomly chosen address of the RaTTle, we effectively index the RaTTle by an offset of 4 (see Figure 6).

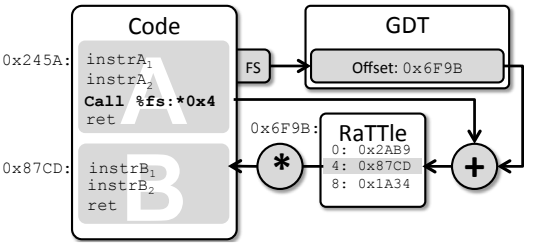


Figure 6: *The RaTTle in Action: Indexed through the GDT and dereferenced using an indirect call; all in* one instruction.

In PALACE code, we substitute each branch and jump instruction with an `%fs` segment override and a unique index. When not using the FS segment override, code does not have access to the RaTTle because it uses a different segment. The address of a segment, and hence of the RaTTle, cannot be read from user space because the local and global descriptor tables point to kernel space memory which is inaccessible from user space. This makes the address of the RaTTle inaccessible.

As a segment selector for the RaTTle, we chose the general purpose segment selector register FS, as already used in the example above. To the best of our knowledge, this register is unused. The only use we found is in the Windows emulator Wine that uses segmentation for its 16-bit Windows emulation.

Efficient Data Access. Data can be accessed in a similar way, but through the Global Offset Table (GOT). The GOT is used in position-independent code such as libraries anyway. We just need to substitute the way the address of the GOT is calculated with an indirection through the RaTTle. Further access is done through the GOT as in traditional position-independent code. This is explained in more detail in Section 4.5.

Populating the RaTTle. The RaTTle is the only part of the code that needs rewriting at load time. The RaTTle is empty in the ELF executable file on disk and its memory gets initialized by the loader with the help of relocation information. This relocation information points to the actual symbols that each RaTTle index refers to. The Linux loader automatically takes the relocation information to rewrite the RaTTle at program load [6].

4 Design Details

With the ingredients described earlier, we can put together our mitigation against code reuse attacks that is efficient, lightweight and shares code and data between processes.

4.1 Design Decisions

There are several ways to implement PALACE. A PALACE executable can be produced by a compiler, or it can be transformed from a traditional executable using static or load-time translation.

Compiler Support. The same way contemporary compilers support PIC, they can be augmented to emit PALACE code. Based on the principles of PALACE code introduced in the previous *Idea* section, the compiler needs to generate PALACE code and put it in subsequent memory-page-sized chunks. It is then ready to be loaded by a traditional loader that permutes the chunks prior to execution of the code.

Static Translation. If the source is not available, an existing executable can be transformed to PALACE by means of static translation [14, 15]. Static translation reads an executable or shared library file from disk, disassembles it, transforms the instructions, and writes a modified executable file back to disk. In our scenario, static translation keeps most of the instructions untouched while only replacing code and data references with the appropriate indirection through the RaTTle.

Load-time Translation. Load-time Translation can be regarded as a static translation that happens automatically at very load-time, after the executable or library has been read from disk into memory but before it starts execution. This method is often referred to as *binary rewrit-*

ing. Its advantage is that a process can be randomized at every startup. In our scenario, however, we do not need load-time translation as we can achieve a randomization at load-time with the specially crafted PALACE chunks in the executable file.

Our Choice. We want to stress that Oxymoron can be implemented by a compiler that simply emits PALACE code in the first place instead of traditional code. We could have implemented Oxymoron as a compiler solution. However, this would have required us to modify existing compilers. Instead, we built a legacy-compatible solution that uses static translation and can be built on an existing fine-grained memory randomization framework, which already uses static translation. We built Oxymoron on the existing framework Xifer provided by Davi et al. [13].

In theory, a static translation approach may seem fragile because it needs a perfect disassembly. However, static translation can be tuned to reliably disassemble code generated by a particular compiler with known and carefully chosen parameters. Besides, in this paper we use the translation from traditional x86 code to PALACE code as a comprehensible running example that demonstrates how PALACE code looks in contrast to traditional x86 code.

In both cases, compiler and static translation, the generated PALACE code of the executables and libraries can be read by a commodity Linux. The Linux OS loader will detect the executable as being ASLR-enabled and will randomize its base address. Unfortunately the commodity loader does not randomize the program segments individually but keeps their relative distances. For traditional position-independent code that was necessary so that code in the `.text` section can still reference objects in the `.data` section by their relative distance to the current instruction pointer. However, for PALACE this limitation is not required. We want to achieve a more fine-grained randomization by allowing an individual randomization of each program segment, which could be as small as a memory page. This can be achieved by requesting a special linker in the program header, which randomizes the segments individually.

4.2 Setting up the RaTTle

The RaTTle needs to be populated with all references in the executable and the table needs to be loaded at a random address. Moreover, one table does not suffice for the interaction of several shared libraries. Before we can use PALACE code, we need to set up the RaTTle as follows:

1. Assign every reference in code a unique number that will act as an index into the RaTTle,
2. Fill the RaTTle with the actual, current, random addresses of the original targets, and
3. Set up segmentation so that a free segment selector points to the RaTTle and we can index the RaTTle.

In step 1, the absolute addresses of the original program are saved in a hash set. Then, every address is assigned an ascending index. This ensures that the table does not grow unnecessarily large. Because the final, random addresses are unknown before the process is started, the RaTTle cannot be filled until start-up of the process. As we want to avoid modification of the operating system loader, we chose a method that is able to fill the RaTTle using only traditional features of the loader. Such a feature is relocation. Relocation information tells the loader which objects in the executable file or in the library must be overwritten with current addresses at load time. Therefore, we add relocation information for each RaTTle index to the final executable/library file. This ensures that the loader rewrites each index so that it points to the corresponding position of code or data that this index represents. As a result, the randomized addresses of the code pieces are automatically written into the RaTTle by the operating system loader.

4.3 Setting up Segmentation

In order to find the RaTTle in memory, we need to set up segmentation so that a pre-defined segment points to the beginning of the table. Unfortunately, we cannot use relocation information for this purpose, because neither setting up segmentation nor setting segment selectors is supported by relocation information. Setting up segmentation via the Global Descriptor Table (GDT) would require kernel modifications. Since the goal is to avoid operating system modifications in order to stay legacy compatible, this is not an option. Luckily, the x86 architecture additionally supports a so-called *Local Descriptor Table* (LDT). The LDT can be switched for every address space, so that Linux emulates a per-process LDT. This is a perfect feature for enabling Oxymoron on a per-process basis.

The set-up of the LDT and the segment selector that points into the LDT is done in initialization code. To this end, we leverage the ELF executable format's initialization code that resides in the `.init` section. Code in this section is ensures to be executed before any other code. This init code figures out the address at which the RaTTle has been randomly loaded by the loader and sets up the LDT accordingly. After the initialization code has run, the segment selector `FS` points to the random address

of the RaTTle. The PALACE code can now work as intended.

4.4 Control Flow and Data

Code. Control flow branches or function calls that target another memory page need to be replaced with an indirection through the RaTTle. The simplest case is a direct `call` or an unconditional `jmp` to a different place in code:

Address	Before	After
8050512:	`call 0x8050c08`	`call %fs:4`
RaTTle:		[0] ..... [4] `0x8050c08`

Only branches that reference code outside of the current memory page must go through the RaTTle. Code and data access within one memory page may be encoded position-relative (e.g., `call +90`).

If the to-be-replaced instruction is an indirect jump, the translation is slightly larger due to the fact that x86 does not support two levels of indirection. It is either possible to use the RaTTle to get the address of the second indirection and then dereference that using an indirect jump or to use a trampoline. We use a trampoline because it is slightly faster:

Address	Before	After
8050512:	`jmp *0x80a00012`	`jmp %fs:4`
80a00012:	`8050c08`	`8050c08`
RaTTle:		[0] ..... [4] `jmp *80a00012`

A slightly more involved case is a conditional jump because there is no equivalent conditional indirect jump. Our solution is a bit more involved:

Address	Before	After
8050512:	`cmp %eax, %ebx`	`cmp %eax, %ebx`
8050514:	`jne 0x8050590`	`jne 0x8050518`
8050516:		`jmp 0x805051a`
8050518:		`jmp *%fs:4`
RaTTle:		[0] ..... [4] `0x8050590`

An indirect jump, such as `jmp *%eax` does not need to be replaced at all. However, the used register (in this example `%eax`) must point to the correct randomized position in memory. This is either ensured by the compiler that generated PALACE code or by the translation from traditional code. In either case, a register is loaded with a code address. Optionally, this address is modified to mimic jump tables or C++ vTables, and then the indirect jump transfers control flow to the address stored in the register. To load a code address to the register before it is modified, a fixed address is copied to the register. This is similar to `mov $0x8402dbc, %eax`. In the case of PALACE, this step needs an indirection to conceal the actual address and to make the address exchangeable by the RaTTle. In PALACE code this register loading looks like this: `mov %fs:$0x4, %eax`. This copies an address stored as an entry in the RaTTle to the register `%eax`. Then, some mathematical operations can be performed, such as adding the offset into C++ vTables and finally the indirect jump is performed as in traditional x86: `jmp *%eax`.

Data Access. Accessing data through the RaTTle is done in exactly the same way. An indirect memory operation is used to read or write data from or to an address stored in the RaTTle. `mov %fs:$0x4, %ebx` is used to read the first entry (4 bytes) of the RaTTle into register `%ebx` and vice versa the operation `mov %ebx, %fs:$0x8` copies the register `%ebx` to the second entry (8 bytes) of the RaTTle.

4.5 Inter-Library Calls and Data

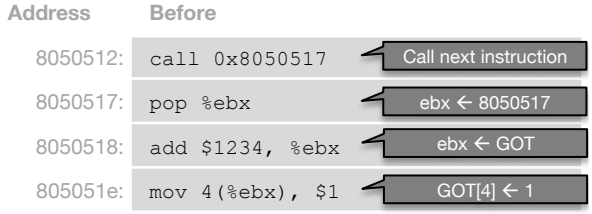
Control flow and access to data is not restricted to one library or executable. Naturally, these code elements frequently use each other's functions and data. Some operating systems, like Windows, use relocation information to directly patch the control flow so that it points into a library after it has been loaded. Linux, on the other hand, uses the procedure linkage table (PLT) to link calls to libraries with the advantage of lazy loading.[1] In contrast, we use an indirection through the RaTTle for every library call or access to global library data because this approach conceals the actual address of the loaded library and has only minor performance impact.

Inter-Library Data. Libraries can export data to be used by the executable main process or other shared libraries. Since it is known a priori which data is accessed in another library, each reference gets a place-holder in the GOT which can be accessed as described above. When the appropriate library is loaded by the loader, it automatically updates the GOT thanks to the relocation info pointing to this entry in the GOT.

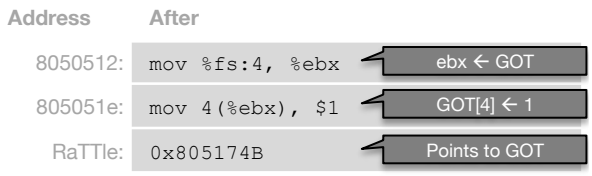
The following is an example of typical position-independent code that uses a GOT to access data: The code is first calling the next instruction, thereby pushing its own address as a return address to the stack. Following, this very address is popped off the stack to get the

[1] First, the PLT entries do not point to the actual procedure inside a library because it has not been loaded yet. Instead, they point to code that loads the library and then rewrites the PLT to link the call to the actual target procedure.

absolute address of the currently running code. The address of the GOT is calculated by adding a known offset.



When transforming this piece of code to PALACE, only the calculation of the GOT needs to be substituted. In this case, the three former instructions get compressed to a single instruction with segment override. Interestingly, this is a faster method of accessing the GOT than the currently used PC-relative addressing.



Inter-Library Calls. Inter-library calls are calls from one loaded library to another or from the main executable to a library. In theory, these calls are no different from a call within the same library or executable because the RaTTle can simply point to code in another library. However, in practice, this would require the RaTTle to reflect all possible combinations of loaded libraries. Therefore, we resort to a solution in which every loaded library brings its own RaTTle and an inter-library call acts as a trampoline that changes the segment selector `FS` to point to the corresponding RaTTle of another library prior to jumping into that library (see in Figure 7).

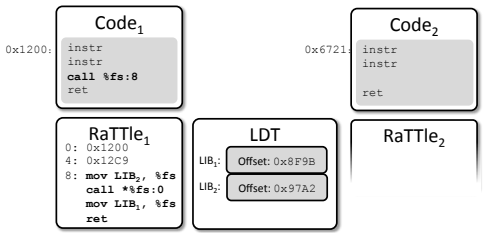


Figure 7: *Inter-Library Calls: Because the indices overlap, a new RaTTle needs to be set up before those calls.*

Please note the missing "*" in the `call %fs:8` of Figure 7, which means the RaTTle is not de-referenced rather than used as a trampoline. This trampoline then lets `FS` point to the index of the other library's RaTTle without the need to know the exact address. Suppose the function that we want to call is stored at index 0 in RaTTle_2, but RaTTle_1 is currently active. The code in Figure 7 first sets `FS` to point to RaTTle_2. RaTTle_2 is the second selector in the LDT. Hence, the trampoline code in RaTTle_1 assigns $10111_{bin} = 23$ to `FS`, which corresponds to a segment selector of "2" (see Appendix A). The trampoline code then jumps to index 0, which now corresponds to currently active RaTTle_2. Because the trampoline uses a `call` instruction to finally call into the other library, control flow returns to the trampoline where `FS` is restored to its former value.

4.6 Debugging

Debugging information augments the executable or library file with annotations describing which memory addresses correspond to which variables or lines of code. These stored addresses must be compatible with Oxymoron randomized addresses. Since Oxymoron is implemented as a static translation tool, the original debugging information needs to be translated as well. Currently Oxymoron supports the common DWARF [3] file format which can be read by the `gdb` or other debuggers. This way, it is possible to teach `gdb` the randomized addresses so that `gdb` can still step through the code, inspect variables etc. like for the non-randomized executable.

5 Evaluation

In this section, we evaluate the effectiveness of Oxymoron empirically as well as theoretically. In order to demonstrate the efficiency, we used the de-facto standard performance benchmark SPEC CPU2006 as well as micro benchmarks to measure cache hit/miss effects.

First, we inspect the security of the RaTTle itself to verify that it did not open the flood gates for other attack vectors. Then, we compare the slightly different randomization of memory pages that this solution entails to the more classical memory randomization solutions in order to get an understanding of the implied security.

5.1 Practical Security Evaluation

We tested our randomization solution against real-life vulnerabilities and exploits. The documented vulnerabilities `CVE-2013-0249` and `CVE-2008-2950` both allow arbitrary code execution by means of return-oriented programming [2]. `CVE-2013-0249` targets the `libcurl` library which handles web requests and is used in dozens of popular programs, including ClamAntiVirus, LibreOffice, and the Git versioning system. The exploit for this vulnerability is crafted in such a way that it triggers a buffer overflow in `libcurl` with the ability to overwrite a return address and ultimately execute a chain of ROP gadgets. The severity of this bug lies in the fact that it can be triggered remotely when `libcurl` accesses

a prepared resource that is under the control of the adversary. In order to test the exploit, we used the 'curl' downloader executable in version 7.28.1, which internally uses `libcurl`. We could successfully run arbitrary code by assembling ROP gadgets at our discretion. After curl had been rewritten to use Oxymoron, the exploit was no longer possible as the addresses that are needed to successfully mount the attack are unknown due to the randomization at every program start.

Similarly, the vulnerability `CVE-2008-2950` allows for arbitrary code reuse in the PDF library `poppler`, which is used by many popular programs such as LibreOffice, Evince and Inkscape. A specially prepared PDF file can trigger an arbitrary memory reference in the `poppler` library, ultimately leading to a code reuse attack. After our attacks against `pdftotext` using `libpoppler` 0.8.4 were successful, we applied Oxymoron. Since the memory address of the PALACE-protected process were no longer known, the exploit was rendered unsuccessful after applying Oxymoron to the pdftotext executable.

5.2 Security of the RaTTle

Because processes are protected by $W \oplus X$ (stack execution prevention), no code can be injected by an attacker. Hence, the only possibility is to reuse existing code. This existing (PALACE) code is littered with `%fs`-prefixed instructions that implicitly point to the RaTTle due to the sheer fact they incorporate a reference to `%fs`. However, the situation is identical to finding ROP gadgets in a classical program, as an attacker needs to know their randomized position in memory in order to chain them together. The fact that this address is not known to an attacker prevents the reuse of code. In fact, the probability of guessing a correct address is negligible (see subsection *"Theoretical Security Evaluation"*).

The RaTTle holds lots of random addresses and, at first glance, seems like a valuable target for an attacker. The security of the RaTTle originates from the fact that its address is unknown and that its content cannot be accessed. All `%fs`-instructions are mere replacements for control flow branches and as such only use the RaTTle as a layer of indirection without ever knowing the actual address of the landing position. If an `%fs`-instruction is a replacement for data access, the same holds true: The RaTTle is only used for indirect access of the actual data. In general, the x86 architecture does not support revealing addresses that segments point to. The only way to read the address is to parse the GDT or LDT which both reside in kernel space. To access the LDT, a user mode program needs to issue a special syscall. Even if a program would consist of ROP gadgets to issue this syscall, he would still need to know the addresses of the required ROP gadgets. So this can be reduced to finding special instructions that can be used as ROP gadgets. This has a negligible probability as explained in *"Theoretical Security Evaluation"*.

5.3 Enhanced Security of the RaTTle

It is possible to further enhance the security of the RaTTle by making it completely inaccessible. The segmentation principle of the x86 architecture allows to distinguish code access from data access. This way, it is possible to set up two different RaTTles, one for code going through `%fs` and one for data going through `%gs`. First of all, in a program without self-modifying code, there should be no instructions that read data using the `%fs` code segment selector. Even if there were, the processor would prohibit such access. Further, it is possible to move the RaTTle completely outside of the normal, otherwise flat[2] data segment (`%ds`). This results in the inability for code to ever access the RaTTle without using proper segment selectors, because it no longer resides in the accessible segment. This is an effective protection against leakage and disclosure attacks (see subsection *"Disclosure Attacks"*). Also, the call stack could be protected using this method. If return addresses are not saved on the regular stack, but rather on a side stack in a reserved area inside the RaTTle, there is no way for memory disclosure vulnerabilities to ever read return addresses and thus they cannot gain information about function addresses.

5.4 Theoretical Security Evaluation

In this subsection we elaborate on why the entropy of memory page granularity randomization is still sufficient for fine-grained randomization and why it is much higher than traditional ASLR.

First, we show that the entropy induced by a page-granular randomization is high enough in the sense that the adversary has only negligible probability of successfully guessing an address. We model the adversary's goal as mounting a code reuse attack against a running program consisting of the executable and its loaded libraries. Hence, his goal is to know the address of either a particular function f of interest (return-into-libc attack) or of several particular instructions $i_1 \dots i_k$ to build gadgets from (ROP attack). Since the contents of a memory page can be extracted from the executable file, the attacker can determine in which memory page the instruction in question resides. Therefore, the success of the adversary re-

[2] A flat segment is a segment that covers the entire address space, i.e. `0x00000000` to `0xFFFFFFFF` on a 32-bit system. This is the default for Windows, Linux and MacOS.

lies on the probability of knowing the address of a particular memory page.

Every memory page is assigned a random address at load-time. Thus, the first page can choose 1 out of n possible page-aligned address slots. The second 1 out of $n-1$ and so forth. For p total process pages to lay out in memory, this yields a total of $\frac{n!}{(n-p)!}$ combinations. The adversary's probability of correctly guessing one address is hence the reciprocal $\frac{(n-p)!}{n!}$. In a 32 bit address space, we have $n = 2^{19} = 524,288$ possible page addresses. The probability of guessing one page correctly therefore is 2^{-19}. That scenario is intuitively identical to ASLR which only randomizes the base address of the code. However, when finding ROP gadget chains, the page granularity drastically lowers the chance of success compared to ASLR because several pages have to be guessed correctly. For a 128 kB ($p = 32$ pages) executable to lay out in memory, the adversary's probability of guessing the correct memory layout therefore is:

$$Pr\left[Adv_{layout}\right] = \frac{(n-p)!}{n!} = \frac{(2^{19}-2^5)!}{2^{19}!} = 2^{-608}$$

Leakage Attacks in ASLR. A leakage vulnerability inadvertently reveals a valid, current address inside the running program. If the adversary additionally knows which object or function has been leaked, he knows the address of that object/function. In the case of ASLR, he can then infer the current addresses of all other objects or functions because ASLR has shifted the entire code segment in memory by changing its base address. Consequently, the relative distances between functions stay exactly the same.

To model the leakage attack, we assume the adversary exploits an existing leakage vulnerability thereby learning a valid address. We assume that this address depicts the beginning of a particular function that the adversary knows. That such a leaked address actually constitutes a function pointer is not very likely but here it models the best-case scenario for the adversary. Hence, the following calculations give a upper bound of success for an adversary.

More formally, the adversary has access to an oracle that can tell which function f has leaked and the adversary can use the leakage vulnerability to learn the current address of A_f of the function f. The adversary can then calculate their difference in memory by calculating their difference in the executable file. As their relative positions did not change in ASLR, the adversary can infer the current address of f' by calculating the difference to the leaked function f. In the case of traditional ASLR, the address of any function f' can be calculated with probability 1. Ultimately, the success probability of the adversary entirely depends on the likelihood of finding such a leakage vulnerability.

Leakage Attacks in Oxymoron. In our case of memory page granularity shuffling, the relative distance between functions varies in general since the code segment is not just shifted en bloc. For any leaked pointer f, there is a chance that it resides in the same memory pages as the desired function f'. For an equal distribution of f' in p pages, the likelihood of f' being in the same page as f is $\frac{1}{p}$. For a program of a total size of only one memory page (4kB), both functions f and f' must reside in the same memory page. Under the assumption that both functions are uniformly distributed, the probability for both to appear in the same memory page is $\frac{1}{p}$ for a program size of p pages. Hence

$$Pr[Adv^{PALACE}_{ret2libc}] \leq \frac{1}{p} \quad \text{and} \quad Pr[Adv^{PALACE}_{ROP}] \leq \frac{1}{p^k}$$

Disclosure Attack. We distinguish between a leakage and a disclosure vulnerability. A disclosure vulnerability allows an attacker to read arbitrary memory content given its address. Snow et al. proposed just-in-time code reuse, which showed that a disclosure vulnerability can significantly reduce the security of fine-grained memory randomization [26]. Just-in-time code reuse repeatedly exploits a memory disclosure vulnerability to map portions of a process' address space with the objective of reusing the so-discovered code in a malicious way. In a fine-grained randomization, the memory pages are scattered across the address space and scanning with arbitrary memory addresses is very likely to end up in unmapped memory. In order not to trap into unmapped memory, they rely on a leakage attack to learn a valid address and then start from this address by disassembling the code in order to follow control flow instructions. Even fine-grained randomization can be reversed using their technique by transitively following the control flow.

However, in our setting of PALACE code, no control flow branch can be followed by reading memory as such a branch only constitutes an offsets into the RaTTle. In order to resolve branches such as `call *%fs:4`, the attacker would need to know the address of the RaTTle or `%fs`, which is not possible, as alluded to earlier. The only chance an attacker has is to rely on a leakage vulnerability to get a valid address. If that address points to data it is useless to the attacker. If it points to code, the attacker can only use a disclosure vulnerability to get the contents of up to a whole memory page (4KB). Otherwise, he is likely to overrun the page and end up in unmapped memory which triggers a page fault that kills the program.

5.5 Effectiveness of Memory Page Sharing

To have a set basic programs one would typically find on a Linux machine, we used the *busybox* project, which incorporates 298 standard Linux commands. Those command line programs were started and their memory footprint was measured using `/proc/<PID>/maps`. On average, they mapped 14.9% more code pages than their unmodified original. Their data pages were unmodified. Only the RaTTle consumes memory (see Subsection 6.1). Compared to fine-grained memory randomization solutions that impede code page sharing, Oxymoron on average saves about 85% of program memory.

6 Performance Evaluation

To evaluate the efficiency of Oxymoron, we did not only use standard command line tools from busybox but conducted CPU benchmarks with PALACE-enabled programs using the de facto standard SPEC CPU2006 integer benchmark suite. All benchmarks were performed on an Intel Core i7-2600 CPU running at 3.4 GHz with 8 GB of RAM.

Static Translation Overhead. Before the executable and libraries can be shuffled in memory, they either need to be compiled with an PALACE-enabled compiler or they must be converted using static translation (cf. section 3). Even though the translation only needs to be performed once, it must be efficient. We measured the rewriting time for all benchmark programs of the Spec CPU suite. The rewriting process is not exactly linear, but on average achieves between 35,000 and 700,000 instructions per second. An overview of the timings of several programs is given in Table 1.

Benchmark	Total # of Instructions	Rewriting Time (s)
483.xalancbmk	1,111,779	4.321
403.gcc	942,244	3.667
471.omnetpp	238,978	0.316
400.perlbench	322,084	1.084
445.gobmk	226,661	6.744
464.h264ref	170,942	0.396
456.hmmer	54,582	0.116
458.sjeng	40,438	0.101
473.astar	32,502	0.032
401.bzip2	28,087	0.056
462.libquantum	15,788	0.024
429.mcf	12,268	0.024

Table 1: *Timings for static rewriting that needs to be done at least once. The total # of instructions include the executable and all its shared libraries.*

The number of instructions per benchmark reflect the total number of instructions from the executable file itself plus its dependent libraries. Note, that this measurement rewrites the entire C-library and other dependent libraries again for each benchmark and is hence slower than just translating the main executable.

Run-Time Overhead. The run-time overhead introduced by the translation through the RaTTle as well as the introduction of `jmp` instructions to connect pages (cf. section 3) is measured in Figure 8. The average run-time overhead of all benchmarks is only 2.7% for the PALACE code and 0.1% for the additionally needed chunking in memory page-sized pieces (4096 bytes).

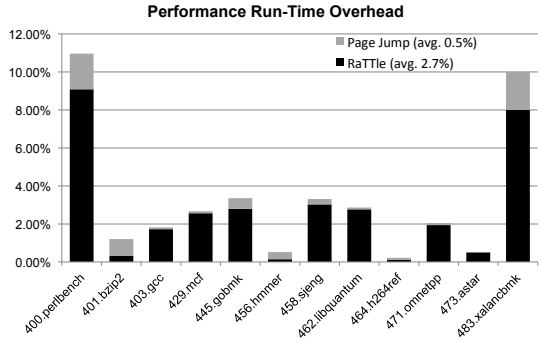


Figure 8: *SPEC CPU2006 integer benchmark results.*

Cache Miss Penalty. We also evaluated the cache effects of our randomization. This is important, since modern processors assume locality of code, which might be thwarted by wild jumping in the code due to the randomization. Keeping cache effects in mind, our implementation optimizes jumping behavior in order to optimize performance under real-life conditions. Our cache experiments showed that PALACE and the randomization have no measurable cache effect.

For this impact to be measured, we handcrafted code consisting of interdependent `add` instructions with a total length of one L1 cache line. These instructions are aligned in memory in such a way that they start at the beginning of a cache line and re-occur such that every cache set and every cache line is filled after execution. We inserted equidistant `jmp` instructions and measured the overhead of 100,000 runs on an Intel Core i7-2600 (32 KB L1 cache, 64 bytes per line). Our results show that the performance impact is not measurable up to every seventh instruction being a `jmp`. If every sixth instruction is a `jmp`, a negligible overhead of 0.4% is introduced. Our analysis of the `busybox` code showed that after translating it to PALACE, on average indeed every 6^{th} instruction was a branch or jump.

6.1 Memory and Instruction Overhead

Compared to a traditional program, the introduction of PALACE code replaced control flow branches with other, `%fs`-relative, instructions. For all SPEC2006 benchmark executables, on average 9% +-1.7% of all instructions are calls that needed to be replaced by indirections through the RaTTle. GOT indirect calls through the RaTTle are only 0.03% of all instructions.

Additionally, a PALACE binary executable file is slightly larger than a traditional executable file because each code page (4 KB) is a separate ASLR-enabled section in the executable file.

During run-time, the memory footprint also slightly increases because the RaTTle has to be kept in memory. Of course, this run-time memory usage is accompanied with the achieved goal of memory savings due to the sharing of code pages with other processes.

Encapsulating each memory page in a separate segment in the ELF file requires the allocation of one section header and one program header per page. A section header is 40 bytes and the ELF program header is 32 bytes which leads to an overhead of 72 bytes per 4096 byte memory page, or $\approx 1.76\%$. Figure 9 depicts both the increase of instructions due the static translation as well as the increase of the ELF section and program headers.

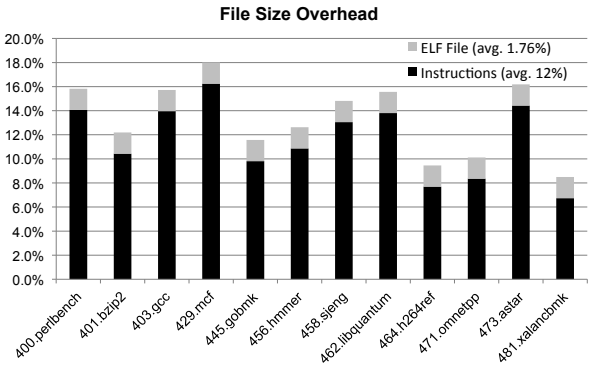


Figure 9: *Memory overhead after static translation.*

Run-Time. The size of the RaTTle depends on how many references the code has. If a target is referenced more then once, e.g., the GOT, only one index is saved in the RaTTle. For all files that belong to the SPECint CPU2006 benchmark, on average 19% of the code segment had to be added in the form of a RaTTle.

7 Related Work

Over the course of the last several years, code reuse attacks and their mitigation has been an ongoing cat and mouse game. Some of the code reuse mitigation techniques address the problem at its roots by preventing buffer overruns or by confining the control flow to the destined control-flow graph. Other mitigation techniques make it hard for the adversary to guess or brute-force addresses that are necessary for successful execution of malicious code.

In this section, we focus on approaches that use fine-grained memory randomization as a means to mitigate code reuse attacks and work that nullifies memory randomization or even fine-grained memory randomization.

One way to categorize fine-grained memory randomization solutions is by their implementation: There exist compiler-based solutions, static or load-time translations, and dynamic translations. Another category dimension is whether they randomize only once, every time the program starts, or even continuously during program execution.

Compiler-Based Solutions. If a program is not randomized, an adversary can learn the layout, i.e. addresses, of all functions and gadgets and hence use them in a ret2libc or ROP attack. The idea of compiler-based approaches is to randomize the layout of a program and to install differently randomized copies on different computers so that the program layout is not predictable for an adversary.

Cohen et al. [10] suggested compiling different versions of the same program. In a modern setting this technique can be applied within an AppStore to distribute individually randomized software. Similarly, Franz et al. [16, 19] have suggested automating this compiler process and generate a different version of a program for every customer. The authors suggest that app store providers integrate a multicompiler in the code production process. However, those approaches have several shortcomings: First, app store providers have no access to the app source code. This requires the multicompiler to be deployed on the developer side, who has to deliver possibly millions[3] of app copies to the app store. Second, the proposed scheme requires software update processes to correctly patch app instances that in turn differ from each other. Finally, the most severe drawback of compiler-based solutions is the fact that the diversified program remains unchanged until an update is provided, which increases the chance of an adversary compromising this particular instance over time.

[3] According to Gartner [4], the number of app downloads is about 102 billion in 2013.

Similar to Oxymoron is the idea of using a compiler-based solution to divide a shared library into even more fragments. Code Islands [30] follows this path and compiles groups of functions to several shared libraries instead of one shared library containing all the functions. These (potentially thousands of shared library files) are then put in a container whose format is understood by a modified loader which maps the libraries in the particular process. However, their solution needs a modified loader to support the proprietary format. Executables then need to load literally thousands of shared libraries, while each library constitutes a single function.

In contrast, Bhatkar et al. [8] presented a source code transformer and its implementation for x86/Linux. The main idea is to augment any source code with the capability of self-diversification for each run. In particular, features are added to the source code that allow the program to re-order its functions in memory in order to mitigate code reuse attacks. Their tool can also be applied to shared libraries if their source code is available. However, their solution induces a run-time overhead of 11% and apparently needs access to the source code.

Static Translation. Static translation reads an executable or shared library file from disk, disassembles it and transforms the instructions according to a predefined pattern within the executable file itself. Kil et al. [20] use static translation for their Address Space Layout Permutation (ASLP). ASLP performs function permutation without requiring access to source code. The proposed scheme statically rewrites ELF executables to permute all functions and data objects of an application. The presented scheme is efficient and also supports re-diversification for each run. However, only the functions themselves are permuted, not their content.

Pappas et al. proposed randomizing instructions and registers within a basic block to mitigate return-oriented programming attacks [21]. However, the proposed solution cannot prevent return-into-libc attacks (which have been shown to be Turing-complete [27]), since all functions remain at their original position.

Load-Time Translation. Load-time translation solutions are similar to static translation but apply the translation at load time in order for the processes to benefit from a re-randomization at each run. This can be achieved by several means, such as rewriting the binary file after it has been loaded but before execution [29, 13]. Such solutions usually suffer from the fact that each execution either needs a translation/rewriting phase each time a process is started or they need a prior static analysis phase [29].

Dynamic Translation. Dynamic translation leaves the original file untouched and does not apply binary rewriting but the program undergoes a *dynamic translation*, i.e. the instructions are transformed as they are executed. Dynamic translation is very similar to Just-in-Time (JIT) compilation but usually translates from and to the same instruction set architecture. For example, Bruening proposed the DynamoRIO framework in his PhD thesis [9]. DynamoRIO is able to perform run-time code manipulation. ILR (instruction location randomization) [18] randomizes the location of each single instruction in the virtual address space. For this, a program needs to be analyzed and re-assembled during a static analysis phase. This is why ILR induces a significant performance overhead (on average 13%), and suffers from a high space overhead, i.e., the rewriting rules reserve on average 104 MB for only one benchmark of the SPEC CPU benchmark suite. For direct calls, ILR can only randomize the return address in 58% of the calls, meaning that for a large number of return instructions, ILR needs to do a live translation for un-randomized return addresses to runtime addresses.

Constant Re-Randomization. To the best of our knowledge, there are only two papers that actually implemented and benchmarked re-randomization. Curtsinger et al. [11] have implemented an LLVM compiler modification that injects code, which adds the functionality to re-randomize the address of functions every 500 ms. According to [11], their overhead of code, heap and stack (re-)randomization is 7%.

Giuffrida et al. [17] changed the Minix microkernel to re-randomize itself every x seconds. This is achieved by maintaining the intermediate language of the LLVM compiler for the compiled kernel modules. However, this procedure has a significant run-time overhead of 10% for a randomization every $x = 5$ seconds or even 50% overhead when applied every second.

Common Shortcomings and Nullification. All the related work on fine-grained memory randomization has in common that they either do not randomize shared libraries, or if they do, the difference introduced in the shared libraries prohibits code sharing.

Furthermore, it is unclear whether fine-grained memory randomization alone is enough to protect against code reuse attacks. Recently, Snow et al. [26] showed that given a memory disclosure vulnerability it is possible to assemble ROP gadgets on-demand without knowing the layout or randomization of a process. They explore the address space of the vulnerable process step by step by following the control flow from an arbitrary start position. After they have discovered enough ROP gadgets

they compile the payload so that it incorporates the actual current addresses that were discovered on-site.

Snow et al. also proposed potential mitigations of their own attack. However, the proposed solutions are either very specific to their heap spraying exploitation or are as general and slow as frequent re-randomization of a whole process. The latter is not even secure if the attack takes place between two randomization phases.

To the best of our knowledge, in this paper we present the first solution that addresses both problems: (1) It is secure against the new just-in-time ROP by Snow et al. (2) It profits from code sharing despite secure randomization.

8 Discussion

In this section we would like to discuss the general applicability of Oxymoron but also its limitations.

The PALACE code presented in this paper only relies on segmentation as an additional hardware feature. Hence, Oxymoron also works in virtualized environments. We successfully tested Oxymoron in software and hardware virtual machines as well as on a para-virtualized Linux using the Xen hypervisor.

The solution presented herein was implemented for the 32 bit x86 architecture. While its 64 bit successor has limited supported for segmentation, the necessary offset functionality of `%fs` segment registers is still available. However, in 64 bit mode, segments do no longer support to set a limit, which makes the RaTTle accessible as data if its address is known.

Another interesting avenue that we did not investigate is just-in-time (JIT) compiled code, such as the Java runtime environment. Those JIT-compilers would need to be adapted in order to emit PALACE-enabled code, otherwise the traditional code they emit is not protected.

9 Conclusion

We presented a novel technique for fine-grained memory randomization that still allows sharing of code among processes. This makes fine-grained memory randomization practical as the memory overhead is significantly reduced in contrast to other randomization solutions. Oxymoron is effective, i.e., code reuse attacks can be mitigated, memory leakage vulnerabilities can no longer be used to revert the randomization, and we presented the first solution to be secure against just-in-time code reuse attacks. The randomized addresses are protected by hardware means, which is an unprecedented security level with a run-time overhead of only 2.7%.

An interesting side effect of our PALACE code is that accessing the Global Offset Table (GOT) uses fewer instructions than the state-of-the-art technique of using PC-relative addressing. Maybe our method could be a slightly faster alternative for accessing the GOT.

References

[1] Common Weakness Enumeration – Top Software Vulnerabilities. `http://cwe.mitre.org/top25/index.html`.

[2] Database of Common Security Vulnerabilities and Exposures. `http://cve.mitre.org`.

[3] Dwarf 2.0 debugging format standard. `http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf`.

[4] Gartner Says Mobile App Stores Will See Annual Downloads Reach 102 Billion in 2013. `http://www.gartner.com/newsroom/id/2592315`.

[5] How to hijack the Global Offset Table with pointers for root shells. `http://www.open-security.org/texts/6`.

[6] Executable and Linking Format (ELF). Tool Interface Standards Committee, May 1995.

[7] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005), ACM, pp. 340–353.

[8] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium* (2005), USENIX Association.

[9] BRUENNING, D. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[10] COHEN, F. B. Operating system protection through program evolution. *Computer & Security 12*, 6 (Oct. 1993), 565–584.

[11] CURTSINGER, C., AND BERGER, E. D. Stabilizer: statistically sound performance evaluation. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 219–228.

[12] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NÜRNBERGER, S., AND SADEGHI, A.-R. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[13] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., AND SADEGHI, A.-R. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *8th ACM SIGSAC symposium on Information, computer and communications security (ACM ASIACCS 2013)* (2013), ACM, pp. 299–310.

[14] DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS) 27*, 5 (2005), 882–945.

[15] EUSTACE, A., AND SRIVASTAVA, A. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), USENIX Association, pp. 25–25.

[16] FRANZ, M. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms* (2010), ACM, pp. 7–16.

[17] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 40–40.

[18] HISER, J. D., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy* (2012).

[19] JACKSON, T., SALAMAT, B., HOMESCU, A., MANIVANNAN, K., WAGNER, G., GAL, A., BRUNTHALER, S., WIMMER, C., AND FRANZ, M. Compiler-generated software diversity. In *Moving Target Defense*, vol. 54 of *Advances in Information Security*. Springer New York, 2011, pp. 77–98.

[20] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC* (2006).

[21] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy* (2012).

[22] PAX TEAM. `http://pax.grsecurity.net/`.

[23] PAX TEAM. PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[24] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[25] SHACHAM, H., JIN GOH, E., MODADUGU, N., PFAFF, B., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *ACM Conference on Computer and Communications Security (CCS)* (2004).

[26] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy* (2013).

[27] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection* (2011), Springer-Verlag.

[28] VAN DER VEEN, V., CAVALLARO, L., BOS, H., ET AL. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 86–106.

[29] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[30] XU, H., AND CHAPIN, S. Address-space layout randomization using code islands. In *Journal of Computer Security* (2009), IOS Press, pp. 331–362.

A LDT Selector Bits

The actual value that a segment selector must hold is not merely an index to the GDT/LDT, but is defined by the architecture set as follows:

Bits 15 - 3	Bit 2	Bit 1 - 0
Number of the entry	0=GDT, 1=LDT	Privilege Level

As user mode is in Ring 3, bits 0 and 1 must be set to 11_{bin}. The use of the LDT forces us to set bit 2 to 1_{bin}. The index "0" of the LDT yields a valid value for the segment selector of 111_{bin} or 7 in decimal.

Password Managers: Attacks and Defenses

David Silver Suman Jana Dan Boneh
Stanford University
Eric Chen Collin Jackson
Carnegie Mellon University

Abstract

We study the security of popular password managers and their policies on automatically filling in Web passwords. We examine browser built-in password managers, mobile password managers, and 3rd party managers. We observe significant differences in autofill policies among password managers. Several autofill policies can lead to disastrous consequences where a remote network attacker can extract multiple passwords from the user's password manager without any interaction with the user. We experiment with these attacks and with techniques to enhance the security of password managers. We show that our enhancements can be adopted by existing managers.

1 Introduction

With the proliferation of Web services, ordinary users are setting up authentication credentials with a large number of sites. As a result, users who want to setup different passwords at different sites are driven to use a password manager. Many password managers are available: some are provided by browser vendors as part of the browser, some are provided by third parties, and many are network based where passwords are backed up to the cloud and synced across the user's devices (such as Apple's iCloud Keychain). Given the sensitivity of the data they manage, it is natural to study their security.

All the password managers (PMs) we examined do not expect users to manually enter managed passwords on login pages. Instead they automatically fill-in the username and password fields when the user visits a login page. Third party password managers use browser extensions to support autofill.

In this paper we study the autofill policies of ten popular password managers across four platforms and show that all are too loose in their autofill policies: they autofill the user's password in situations where they should not thereby exposing the user's password to potential attackers. The results can be disastrous: an attacker can extract many passwords from the user's password manager without the user's knowledge or consent as soon as the user connects to a rogue WiFi network such as a rogue router at a coffee shop. Cloud-based password syncing further exacerbates the problem because the attacker can potentially extract user passwords that were never used on the device being attacked.

Our results. We study the security of password managers and propose ways to improve their security.

- We begin with a survey of how ten popular password managers decide when to autofill passwords. Different password managers employ very different autofill policies, exposing their users to different risks.

- Next, we show that many corner cases in autofill policies can lead to significant attacks that enable remote password extraction without the user's knowledge, simply by having the user connect to a rogue router at a coffee shop.

- We believe that password managers can help strengthen credential security rather than harm it. In Section 5 we propose ways to strengthen password managers so that users who use them are more secure than users who type in passwords manually. We implemented the modifications in the Chrome browser and report on their effectiveness.

We conclude with a discussion of related work on password managers.

An example. We give many examples of password extraction in the paper, but as a warm-up we present one example here. Consider web sites that serve a login page over HTTP, but submit the user's password over HTTPS (a setup intended to prevent an eavesdropper from reading the password but actually leaves the site vulnerable). As we show in Section 4, about 17% of the Alexa Top 500 websites use this setup. Suppose a user, Alice, uses a password manager to save her passwords for these sites

At some point later, Alice connects to a rogue WiFi router at a coffee shop. Her browser is directed to a landing page that asks her to agree to the terms of service, as is common in free WiFi hotspots. Unbeknownst to Alice, the landing page (as shown in Figure 1) contains multiple invisible iFrames pointing to the login pages of the websites for which Alice has saved passwords. When the browser loads these iFrames, the rogue router injects JavaScript into each page and extracts the passwords autofilled by the password manager.

This simple attack, without any interaction with the user, can automatically extract passwords from the password manager at a rate of about ten passwords per second. Six of the ten password managers we examined were vulnerable to this attack. From the user's point of view, she simply visited the landing page of a free WiFi hotspot. There is no visual indication that password extraction is taking place.

Figure 1: A sample landing page of a rogue WiFi hotspot containing invisible iFrames to the target sites. Note that the iFrames are actually invisible to the user and shown here only for clarity.

2 Password managers: a survey

We begin with a detailed survey of the autofill policies implemented in widely deployed password managers. The password managers we survey include:

- **Desktop Browser PMs**: Google Chrome 34, Microsoft Internet Explorer 11, Mozilla Firefox 29, and Apple Safari 7.
- **3rd Party PMs**: 1Password [1], LastPass [5], Keeper [7], Norton IdentitySafe [6], and KeePass [4]. All of these besides KeePass provide browser extensions that support password field autofill.
- **iOS PMs**: Mobile Safari's password manager syncs with the desktop version of Safari through Apple's iCloud Keychain synchronization service. Since mobile Safari does not support extensions, 3rd Party PMs are separate applications with their own built-in web browser. In addition to Mobile Safari, we survey password managers in Google Chrome, 1Password, and LastPass Tab.
- **Android PMs**: the default Android browser and Chrome.

All these password managers offer an "autofill" functionality, wherein the password manager automatically populates the username and password fields within the user's web browser. We divide autofill strategies into two broad categories:

- **Automatic autofill:** populate username and password fields as soon as the login page is loaded without requiring any user interaction. Password managers that support automatic autofill include Chrome (all platforms), Firefox, Safari, LastPass, Norton IdentitySafe, and LastPass Tab.
- **Manual autofill:** require some user interaction before autofilling. Types of interaction include clicking on or typing into the username field, pressing a keyboard shortcut, or pressing a button in the browser. Password managers that always require manual interaction include 1Password, Keeper, and KeePass.

Internet Explorer 11 uses a hybrid approach: it automatically autofills passwords on pages loaded over HTTPS, but requires user interaction on pages loaded over HTTP. We show in Section 4 that even this conservative behavior still enables some attacks.

Some password managers require manual interaction for autofill in specific situations:

- Chrome requires manual interaction if the password field is in an iFrame.
- Chrome can read passwords stored in Mac OS X's system-wide keychain, but will not automatically autofill them until they have been manually selected by the user at least once.
- The first time Safari or Chrome on Mac OS X access a password in the system keychain, a system dialog requests permission from the user. If the user chooses "Always Allow", this dialog will not be shown again and the password will automatically autofill in the future. This dialog does not appear if the password was synchronized from another device using iCloud Keychain.
- LastPass and Norton IdentitySafe provide non-default configuration options to disable automatic autofill. In this paper we only discuss the default configurations for these password managers.

Platform	Password manager	Same protocol and action	Different protocol	Different form action on load	Different form action on submit	auto-complete = "off"	Broken HTTPS
Mac OS X 10.9.3	Chrome 34.0.1847.137	Auto	No Fill	Manual	Auto	Auto	No Fill
	Firefox 29.0.1	Auto	No Fill	None	Auto	No Fill	Auto
	Safari 7.0.3	Auto	No Fill	Auto	Auto	Auto	Auto
Safari ext.	1Password 4.4	Manual	Manual	Manual	Manual	Manual	Manual
Safari ext.	LastPass 3.1.21	Auto	Manual	Warning	Auto	Auto	Auto
Safari ext.	Keeper 7.5.26	Manual	Manual	Manual	Manual	Manual	Manual
Windows 8.1 Pro	IE 11.0.9600.16531	Auto/Man	No Fill	Auto/Man	Auto/Man	Auto/Man	Manual
	KeePass 2.24	Manual	Manual	Manual	Manual	Manual	Manual
IE addon	IdentitySafe 2014.7.0.43	Auto	Auto	Auto	Auto	Auto	Auto
iOS 7.1.1	Mobile Safari	Auto	No Fill	Auto	Auto	No Fill	Auto
	1Password 4.5.1	Manual	Manual	Manual	Manual	Manual	Manual
	LastPass Tab 2.0.7	Auto	Manual	Auto	Auto	Auto	Auto
	Chrome 34.0.1847.18	Auto	No Fill	No Fill	Auto	No Fill	Auto
Android 4.3	Chrome 34.0.1847.114	Auto	No Fill	No Fill	Auto	Auto	No Fill
	Android Browser	Auto	No Fill	Auto	Auto	No Fill	Auto

Table 1: Password Manager autofill behavior (automatic autofill, manual autofill, or no fill), depending on the protocol (http/https), autocomplete attribute, form action used on the current page relative to the protocol, and form action used when the password was saved. *Manual autofilling* refers to autofilling a password after some user interaction, such as a click or tap on one of the form fields. *No fill* means that no autofilling of passwords takes place. The second to last column refers to autofill behavior when the password field's autocomplete attribute is set to "off". The last column refers to autofill behavior for a login page loaded over a bad HTTPS connection.

2.1 Autofill policies

Next, we ask what happens when the PM is presented with a login page that is slightly different from the login page at the time the password was saved. Should the PM apply autofill or not? Different PMs behave differently and we survey the different policies we found. Table 1 summarizes some of our findings.

The domain and path. All password managers we tested allow passwords to be autofilled on any page within the same domain as the page from which the password was originally saved. For example, a password originally saved on `https://www.example.com/aaa.php` would be filled on `https://www.example.com/bbb.php`. This allows autofill to function on sites that display the login form on multiple pages, such as in a page header visible on all pages. It also allows autofill after a site redesign that moves the login form.

This feature means that an attacker can attack the password manager (as in Section 4) on the *least-secure* page within the domain. It also means that two sites hosted on the same domain (ie, `example.edu/~jdoe` and `example.edu/~jsmith`) are treated as a single site by the password manager.

Protocol: HTTP vs. HTTPS. Suppose the password was saved on a login page loaded over one protocol (say, HTTPS), but the current login page is loaded over a different protocol (say, HTTP)? All other elements of the URL are the same, including the domain and path. Should the password manager autofill the password on the current login page?

Chrome, Safari, Firefox, and Internet Explorer all refuse to autofill if the protocol on the current login page is different from the protocol at the time the password was saved. However, 1Password, Keeper, and LastPass all allow autofill after user interaction in this case. Note that LastPass normally uses automatic autofill, so this downgrade to manual autofill on a different protocol was implemented as a conscious security measure. Norton IdentitySafe does not pay attention to the protocol. It automatically autofills a password saved under HTTPS on a page served by HTTP. As we show later on, any form of autofilling, manual or not, is dangerous on a protocol change.

Modified form action. A form's action attribute specifies where the form's contents will be sent to upon submission.

```
<form action="example.com" method="post">
```

One way an attacker can steal a user's password is to change the action on the login form to a URL under the

attacker's control. Therefore, one would expect password managers to not autofill a login form if the form's action differs from the action when the password was first saved.

We consider two different cases. First, suppose that at the time the login page is loaded the form's action field points to a different URL than when the password was first saved. Safari, Norton IdentitySafe and IE (on HTTPS pages) nevertheless automatically autofill the password field. Desktop Chrome and IE (on HTTP pages) autofill after some manual interaction with the user. LastPass asks for user confirmation before filling a form whose action points to a different origin than the current page.

Second, suppose that at the time the login page is loaded the form's action field points to the correct URL. However, JavaScript on the page modifies the form action field so that when the form is submitted the data is sent to a different URL. All of the password managers we tested allow an autofilled form to be submitted in this case even though the password is being sent to the wrong location. We discuss the implications of this in Section 4 and discuss mitigations in Section 5.

Password managers without automatic autofill require user interaction before filling the form, but none give any indication to the user that the form's action does not match the action when the credentials were first saved. Since a form's action is normally not visible to the user, there is no way for the user to be sure that the form was submitting to the place the user intended.

The effects of the action attribute on autofill behavior is captured in the third and fourth columns of Table 1.

Autocomplete attribute A website can use the *autocomplete* attribute to suggest that autocompletion be disabled for a form input [3]:

```
<input autocomplete="off" ... >
```

We find that Firefox, Mobile Safari, the default Android Browser, and the iOS version of Chrome respect the autocomplete attribute when it is applied to a password input. If a password field has its autocomplete attribute set to "off", these password managers will neither fill it nor offer to save new passwords entered into it. All of the other password managers we tested fill the password anyway, ignoring the value of the autocomplete attribute. LastPass ignores the attribute by default, but provides an option to respect it.

Once the password manager contains a password for a site, the autocomplete attribute does not affect its vulnerability to the attacks presented in this paper. As described in Section 4, in our setting, the attacker controls the network and can modify the login form to turn the password input's autocomplete attribute on even if the victim website turns it off.

In supporting browsers, the autocomplete attribute can be used to prevent the password from being saved at all. This trivially defends against our attacks, as they require a saved password. However, it is not a suitable defense in general due to usability concerns. A password manager that doesn't save or fill passwords will not be popular amongst users.

Broken HTTPS behavior. Suppose the password was saved on a login page loaded over a valid HTTPS connection, but when visiting this login page at a later time the resulting HTTPS session is broken, say due to a bad certificate. The user may choose to ignore the certificate warning and visit the login page regardless. Should the password manager automatically autofill passwords in this case? The desktop and Android versions of Chrome refuse to autofill passwords in this situation. IE downgrades from automatic to manual autofill. All other password managers we tested autofill passwords as normal when the user clicks through HTTPS warnings. As we will see, this can lead to significant attacks.

Modified password field name. All autofilling password managers, except for LastPass, autofill passwords even when the password element on the login page has a `name` that differs from the name present when the password was first saved. Autofilling in such situations can lead to "self-exfiltration" attacks, as discussed in Section 5.2.1. LastPass requires manual interaction before autofilling a password in a field whose name is different from when the password was saved.

2.2 Additional PM Features

Several password managers have the following security features worth mentioning:

iFrame autofill. Norton IdentitySafe, Mobile Safari and LastPass Tab do not autofill a form in an iFrame that is not same-origin to its parent page. Desktop Chrome requires manual interaction to autofill a form in an iFrame regardless of origin. Chrome for iOS and the Android browser will never autofill an iFrame. Firefox, Safari, and Chrome for Android automatically autofill forms in iFrames regardless of origin.

Safari and Mobile Safari will only autofill a single login form per top-level page load. If a page, combined with all of its iFrames, has more than one login form, only the first will be autofilled.

We discuss the impact of these policies on security in Section 4.

Visibility. Norton IdentitySafe does not automatically autofill a form that is invisible because its CSS display attribute is set to "none" (either directly or inherited from a parent). However, it *will* automatically autofill a form with an opacity of 0. Therefore, this defense does not enhance security.

Autofill method. KeePass is unique amongst desktop password managers in that it does not integrate directly with the browser. Instead, it can "autotype" a sequence of keystrokes into whatever text field is active. For most login forms, this means it will type the username, the Tab key, the password, then the Enter key to populate and submit the form.

Autofill and Submit. 1Password, LastPass, Norton IdentitySafe, and KeePass provide variants of "autofill and submit" functionality, in which the password managers not only autofills a login form but also automatically submits it. This frees the user from interacting with the submit button of a login form and thus makes autofill more convenient for the user.

3 Threat Model

In the next section we present a number of attacks against password managers that extract passwords from all the managers we examined. First, we define the attackers capabilities and goals. We only consider active man-in-the-middle network attackers i.e. we assume that the adversary can interpose and modify arbitrary network traffic originating from or destined to the user's machine. However, unlike standard man-in-the-middle attacks, we do not require the user to log into any target websites in the presence of the attacker. Instead, the setup consists of two phases:

First, the user logs in to a number of sites and the attacker cannot observe or interfere with these logins. The user's password manager records the passwords used for these logins. For password managers that support syncing of stored passwords across multiple machines (e.g., Apple's iCloud KeyChain), users may even carry out this step on an altogether different device from the eventual victim device.

At a later time the user connects to a malicious network controlled by the attacker, such as a rogue WiFi router in a coffee shop. The attacker can inject, block, and modify packets and its goal is to extract the passwords stored in the user's password manager without any action from the user.

We call this type of attacker the evil coffee shop attacker. These attacks require only temporary control of a network router and are much easier and thus more likely to happen in practice. We show that even such weak man in the middle attackers can leverage design flaws in password managers to remotely extract stored passwords without the user logging into any website.

The attacker has no software (malware) installed on the user's machine. We only assume the presence of a password manager acting in the context of a web browser.

4 Remote extraction of passwords from password managers

We show that an evil coffee shop attacker can extract passwords stored in the user's password manager. In many of our attacks the user need not interact with the victim web site and is unaware that password extraction is taking place. We discuss defenses in Section 5.

4.1 Sweep attacks

Sweep attacks take advantage of automatic password autofill to steal the credentials for multiple sites at once without the user visiting any of the victim sites. For password managers backed by a syncing service (such as Apple's iCloud Keychain) the attacker can extract site passwords even if the user never visited the site on that device. These attacks work in password managers that support automatic autofill, highlighting the fundamental danger of this feature.

Sweep attacks consist of three steps. First, the attacker makes the user's browser visit an arbitrary vulnerable webpage at the target site without the user's knowledge. Next, by tampering with network traffic the attacker injects JavaScript code into the vulnerable webpage as it is fetched over the network using one of the methods described in Section 4.2. Finally, the JavaScript code exfiltrates passwords to the attacker using the techniques in Section 4.3.

In the sweep attacks we implemented, the user connects to a WiFi hotspot controlled by the attacker. When the user launches the browser, the browser is redirected to a standard hotspot landing page asking for user consent to standard terms of use. This is common behavior for public hotspots. Unbeknownst to the user, however, the landing page contains invisible elements that implement the attack.

iFrame sweep attack. Here the innocuous hotspot landing page contains invisible iFrames pointing to the arbitrary pages at multiple target sites. When the browser loads these iFrames, the attacker uses his control of the router to inject a login form and JavaScript into each iFrame using the methods described in Section 4.2. As we will see, injecting a login form and JavaScript is not

difficult and can be done in several different ways. All that is needed is *some* vulnerable page on the target site. It is especially easy for sites that serve their login page over HTTP (but submit passwords over HTTPS), which is a common setup discussed in the next section.

As each iFrame loads, the password manager will automatically populate the corresponding password field with the user's password. The injected JavaScript in each iFrame can then steal and exfiltrate these credentials.

Our experiments show that this method can extract passwords, unbeknownst to the user, at a rate of about ten passwords per second. To prevent the user from clicking through the landing page before the attacks are done, the landing page includes a JavaScript animated progress bar that forces the user to wait until the attacks complete.

We also find that the password extraction process can be made more efficient by arranging the iFrames in a hierarchical structure instead of adding one iFrame to the top-level page for each target website. Adding all the iFrames to the top-level page would create large increases in both the amount of traffic on the network and the amount of memory used by the victim's browser. Hierarchical arrangement of the iFrames can avoid such issues. The top-level iFrame contains most of the code for the attack and dynamically spawns child frames and navigates them to the target pages. This technique allows the iFrames to load asynchronously and thus ensures that network and memory usage remain reasonable for the duration of the attack.

Chrome (all platforms) is the only automatic autofill password manager that is not vulnerable to the iFrame-based attack, because they never automatically autofill passwords in iFrames. All the other automatic autofill password managers are vulnerable to this attack. Even though the autofill policies of Norton IdentitySafe, Safari, Mobile Safari, and LastPass Tab described in Section 2.2 restrict the number of passwords that can be stolen in a single sweep to 1, they remain vulnerable.

Window sweep attack. A variant of the sweep attack uses windows instead of invisible iFrames. If the attacker can trick users into disabling their popup blocker (e.g., by requiring a window to open before the user can gain access to the WiFi network), the landing page can open each of the victim pages in a separate window. This is more noticeable than the iFrame-based approach, but the JavaScript injected into each victim page can disguise these windows to minimize the chances of detection. Techniques for disguising the windows include minimizing their size, moving them to the edge of the screen, hiding the pages' contents so that they appear to the user as blank windows, and closing them as soon as the password has been stolen.

Nearly all automatic autofill password managers, including desktop Chrome, are vulnerable to the window-based attack. Only LastPass Tab is not vulnerable, as it does not support popup windows at all. Hence, although iFrames make the sweep attack easier, they are not required.

Redirect sweep attack. A redirect sweep attack enables password extraction without any iFrames or separate windows. In our implementation, once the user connects to a network controlled by the attacker and requests an arbitrary page (say, a.com), the network attacker responds with an HTTP redirect to some vulnerable page on the target site (say, b.com). The user's browser receives the redirect and issues a request for the page at b.com. The attacker allows the page to load, but injects a login form and JavaScript into the page, as described in Section 4.2. The injected JavaScript disguises the page (for example, by hiding its body) so that the user does not see that b.com is being visited.

When the user's browser loads the page from b.com, the vulnerable password manager will automatically autofill the login form with the credentials for b.com, which the injected JavaScript can then exfiltrate. Once done, the injected JavaScript redirects the user's browser to the next victim site, (say c.com) and exfiltrates the user's password at c.com in the same way. When sufficiently many passwords have been exfiltrated the attacker redirects the user's browser to the original page requested by the user (a.com).

This attack leaves small indications that password extraction took place. While the attack is underway the user's address bar will display the address of the attacked site, and the attacked site will remain in the user's history. However, as long as the body of the page itself is disguised, most users will not notice these small visual clues.

All of the automatic autofill password managers we tested were vulnerable to this attack.

Summary. Table 2 describes which password managers are vulnerable to these sweep attacks.

Attack amplification via password sync. Most password managers offer services that synchronize users' passwords between different devices. These password synchronization services can potentially result in password extraction from devices without them ever having visited the victim site.

Suppose the user's password manager syncs between their desktop and tablet, and will automatically autofill a password synced from another device without user in-

Platform	Password Manager	iFrame sweep	Window sweep	Redirect sweep
Mac OS X 10.9.3	Chrome 34.0.1847.137		+	+
	Firefox 29.0.1	+	+	+
	Safari 7.0.3	Single	+	+
Safari ext.	1Password 4.4			
Safari ext.	LastPass 3.1.21	+	+	+
Safari ext.	Keeper 7.5.26			
Windows 8.1 Pro	Internet Explorer 11.0.9600.16531	HTTPS	HTTPS	HTTPS
	KeePass 2.24			
IE addon	Norton IdentitySafe 2014.7.0.43	SO	+	+
iOS 7.1.1	Mobile Safari	Single, SO	+	+
	1Password 4.5.1			
	LastPass Tab 2.0.7	SO		+
	Chrome 34.0.1847.18		+	+
Android 4.3	Chrome 34.0.1847.114		+	+
	Android Browser		+	+

Table 2: Vulnerability to sweep attacks. **+** indicates vulnerability without restriction. **HTTPS** indicates vulnerability only on pages served over HTTPS. **Single** indicates a single site is vulnerable per top-level page load. **SO** indicates vulnerability when the page containing the iFrame is same-origin with the target page in the iFrame.

teraction. Suppose further that the site c.com is vulnerable to network attacks and thus to the attacks described above. The user is careful and only ever visits c.com on their desktop, which never leaves the user's safe home network. However, when the user connects their tablet to the attacker's WiFi network at a coffee shop, the attacker can launch a sweep attack on the user's tablet and extract the user's password for c.com even though the user has never visited c.com on their tablet.

We tested Apple's iCloud Keychain, Google Chrome Sync, Firefox Sync, and LastPass Tab, and found all of them to be vulnerable to this attack. In general, any password manager that automatically autofills a password synced from another device will be vulnerable to this type of attack amplification. Therefore, the security of any password manager is only as strong as the security of the weakest password manager it syncs with.

4.2 Injection Techniques

Sweep attacks rely on the attacker's ability to modify a page on the victim site by tampering with network traffic. The attacks are simplest when the vulnerable page is the login page itself. However, any page that is same-origin with login page is sufficient, as all password managers associate saved passwords with domains and ignore the login page's path. The attacker can inject a login form into any page in the origin of the actual login page and launch a password extraction attack against that page. We list a few viable injections techniques.

HTTP login page. Consider a web site that serves its login page over HTTP, but submits the login form over HTTPS. While this setup protects the user's password from eavesdropping when the form is submitted, a coffee shop attacker can easily inject the required JavaScript into the login form at the router and mount all the sweep attacks discussed in the previous section.

Clearly serving a login form over HTTP is bad practice because it exposes the site to SSLstrip attacks [33]. However extracting passwords via SSLstrip requires users to actively enter their passwords while connected to the attacker's network and visiting the victim page. In contrast, the sweep attacks in the previous section extract passwords without any user interaction.

To test the prevalence of this setup — a login page loaded over HTTP, but login form submitted over HTTPS — we surveyed Alexa Top 500 sites (as of October 2013) by manually visiting them and examining their login procedures. Of the 500 sites surveyed, 408 had login forms. 71 of these 408 sites, or 17.40%, use HTTP for loading the login page, but HTTPS for submitting it. Some well known names are on this list of 71 sites, including `ask.com`, `godaddy.com`, `reddit.com`, `huffingtonpost.com`, and `att.com`.

Additionally, 123 (or 30.15%) of the sites used HTTP both for loading the login page and for submitting it. This setup is trivially vulnerable to eavesdropping, but a vulnerable password manager increases this vulnerability by removing the need for a human to enter their password. For the purposes of our attacks, these sites can be thought of as an especially vulnerable subclass of sites with a login form served over HTTP.

Passwords for all these vulnerable websites can be easily extracted from an autofilling password manager using the sweep attacks in the previous section. One could argue that all these sites need to be redesigned to load and submit the login page over HTTPS. However, until that is done there is a need to strengthen password managers to prevent these attacks. We discuss defenses in Section 5.

Embedded devices I. Many embedded devices serve their login pages over HTTP by default because the channel is assumed to be protected by a WiFi encryption protocol such as WPA2. Indeed, Gourdin et al. report that the majority of the embedded web interfaces still use HTTP [26]. Similarly, internal servers in a corporate network may also serve web login pages over HTTP because access to these servers can only be done over a Virtual Private Network (VPN).

Sweep attacks are very effective against these devices: the password manager autofills the password even when the underlying network connection is insecure. By injecting JavaScript into the HTTP login page as above, a coffee shop attacker can extract passwords for embedded devices and corporate servers that the user has previously interacted with.

Embedded Devices II. Some home routers serve their login pages over HTTPS, but use are self-signed certificates. An attacker can purchase a valid certificate for the same common name as the router's [38] or generate its own self signed certificate. When the user's machine connects to the attacker's network, the attacker can spoof the user's home router by presenting a valid certificate for the router's web site. This allows the attacker to mount the sweep attack and extract the user's home router password.

Broken HTTPS. Consider a public site whose login page is served over HTTPS. In Section 2 we noted that many password managers that autofill passwords automatically do so even when the login page is loaded over a broken HTTPS connection, say due to a bad certificate. This can be exploited in our redirect sweep attack: when the browser is redirected to the victim site, the attacker serves the modified login page using a self signed cert for that site. This modified login page contains a login form and the JavaScript needed to exfiltrate the user's password once it is autofilled by the password manager.

These self signed certs will generate HTTPS warning in the browser, but if the redirect sweep attack happens as part of the process of logging on to the hotspot, the user is motivated to click through the resulting HTTPS warning messages. As a result the attacker can extract user passwords from the password manager, even for sites where the login page is served over HTTPS.

Indeed, several prior works have found that users often tend to click through HTTPS warnings [43, 8]. The user may decide to click through the warning and visit the site anyway, but not enter any sensitive information. Nevertheless, the user's password manager autofills the password resulting in password extraction by the attacker, regardless of the user's caution. All of the password managers we tested fill passwords even when the user has clicked through an SSL warning, with the exception of the desktop and Android versions of Chrome.

Active Mixed Content. Any HTTPS webpage containing active content (e.g., scripts) that is fetched over HTTP is also a potential vector. If rendering active mixed content is enabled in the user's browser, any HTTPS page containing active mixed content is vulnerable to injection. Chrome, Firefox, and IE block active mixed content by default but provide a user option to enable it. Safari, Mobile Safari, and the Android stock browser allow active mixed content to be fetched and executed without any warnings. Several types of active mixed content, especially those processed by browser plugins, are harder to block. For example, embedding a Shockwave Flash (SWF) file over HTTP if not blocked correctly can be used by a network attacker to inject arbitrary scripts [30].

XSS Injection. A cross-site scripting vulnerability in a page allows the attacker to inject JavaScript to modify the page as needed [24]. XSS vulnerabilities are listed as one of the most common web vulnerabilities in 2013 internet security threat report by Symantec [20]. If an XSS vulnerability is present on *any* page of the victim site, the sweep attacks will work even if the site's login page is served over HTTPS. For example, the attacker simply includes an iFrame or a redirect on the malicious hotspot landing page that links to the XSS page. The link uses the XSS vulnerability to inject the required login form and JavaScript into the page.

Furthermore, an XSS vulnerability allows for a weaker threat model than our coffee shop attacker. An ordinary web attacker can trick the user into visiting his site, then launch the attack through the XSS vulnerability. This style of attack requires no access to the user's network and has been suggested previously by RSnake [37] and Saltzman et al. [40].

Leftover Passwords. The user's password manager may contain leftover passwords from older, less secure versions of a site. An attacker could spoof the old site to steal the leftover password. Unless the user is proactive about removing older passwords, updating the security of the site does not protect the domain from this type of

attack. For example, if a user's password manager contained a password for Facebook from before its switch to HTTPS, an attacker could spoof an HTTP Facebook login page to steal the password.

4.3 Password Exfiltration

In the previous section we referred to JavaScript that exfiltrates the user's password once it is autofilled by the password manager. Once the password manager has autofilled the login form, the attacker must be able to access the filled-in credentials and send them to a server under its control. We briefly describe two methods for accomplishing this.

4.3.1 Method #1: Stealth

Using stealth exfiltration, the attacker waits until the login form is populated with the user's credentials automatically by a password manager, then steals the password by loading an attacker controlled page in an invisible iFrame and passing the credentials as parameters. The following simple JavaScript does just that and works with all password managers we tested:

```
function testPassword() {
 var password =
        document.forms[0].password.value;
 if(password != "") {
  var temp = document.createElement("div");
  temp.innerHTML +=
        "<iFrame src=\""+ attacker_addr +
        "?password=" + password +
        "\" style=\"display:none;\" />";
  document.body.appendChild(temp);
  clearInterval(interval);
}}
interval = setInterval(testPassword, 50);
```

4.3.2 Method #2: Action

An HTML form's "action" is the URL to which the form's data will be submitted. The attacker can modify a login form's action attribute so that it submits to an attacker-controlled site, thereby leaking the user's credentials to the attacker. If the attacker redirects the user's browser back to the real action, the user will not notice the change.

Automatic autofill password managers populate password forms when the page first loads. The attacker can then use injected JavaScript to change the action, submit the form and steal the password. If the login page is loaded in an iFrame or if it is rendered invisible, the users will not even realize that a login form was submitted. The following simple code does just that:

```
changer = function() {
  document.forms[0].action = attacker_addr;
  document.forms[0].submit();   }
setTimeout(changer, 1000);
```

In section 2.1 we showed that password managers that automatically autofill passwords do so on page load and show no warning to the user when the submitted form action differs from the action when the password was first saved. Thus, all password managers with automatic autofill are vulnerable to this exfiltration method.

4.4 Attacks that need user interaction

All of the attacks described thus far take advantage of automatic autofill password managers to work when the user does not interact with the login form. However, the exfiltration techniques we described work regardless of how the login form was filled. If the user's password manager requires user input to fill passwords and an attacker can trick the user to interact with the login form without them realizing it, the same exfiltration techniques can be used to steal the password as soon as the password form is filled.

We created a simple "clickjacking" attack [29, 39, 31]. The attacker presents the user with a benign form seemingly unrelated to the target site. Overlaying the benign form is an invisible iFrame pointing to the target site's login page. The iFrame is positioned such that when a user interacts with the benign form, they actually interact with the invisible iFrame — in this case, when the user thinks they are filling a form on a benign site, they are actually filling the password in the target site. Once filled, any of the exfiltration techniques described previously can be used to steal the password. This attack steals a password for one site at a time, but could be repeated to steal passwords for multiple sites.

We confirmed this attack works against both Chrome and Internet Explorer 11, as both required manual interaction before filling in at least some situations.

5 Strengthening password managers

In this section we present two complementary solutions to the attacks presented earlier. Before describing the details of our solutions, we first describe why some of the obvious solutions do not work. For example, as all our attacks require JavaScript injection, a potential solution is to prevent password managers from autofilling passwords on a page that is vulnerable to JavaScript injection. This solution is hard to implement in practice as some JavaScript injection vectors (e.g., XSS bugs) are extremely hard for the browser to detect. Another possible solution is to completely block autofill inside iFrames. However, this solution does not prevent the window or redirect sweep attacks described in Section 4.

Moreover, blocking autofill inside iFrames will inconvenience users of benign websites that include login forms inside iFrames.

5.1 Forcing user interaction

Our ultimate goal is to ensure that using a password manager results in better security than when users manually enter passwords in a password field. This is certainly not the case with password managers today, as the attacks of the previous section demonstrate. We begin with the simplest defense that makes password managers no worse than manual user entry.

Our most powerful attacks exploit the automatic autofill of the password field. An obvious defense is to *always* require some user interaction before autofilling a form. This will prevent sweep attacks where multiple passwords are extracted without any user interaction. Interaction can come in the form of a keyboard shortcut, clicking a button, selecting an entry from a menu, or typing into the username field. Regardless of the type of interaction, it must be protected against clickjacking attacks as described in Section 4.4. The user interaction should occur through trusted browser UI that JavaScript cannot interact with, preventing malicious JavaScript from spoofing user interaction and triggering an autofill.

Furthermore, the password manager should show the domain name being autofilled before the filling occurs, so that users know which site is being autofilled. This reduces the chances of the user filling a form without meaning to. For example, if a login page for one site contains an invisible iFrame pointing to the login page of another site, the user must explicitly choose which domain they want filled.

In some settings, such as broken HTTPS, the password manager should simply refuse to autofill passwords.

Implementation. Always forcing user interaction was easy to prototype in Chrome[1] because Chrome already requires user input in certain situations, such as when the action on the current page is different from the action when the password was saved. Since the UI implementation already existed we simply had to always trigger it. We did so by hardcoding the `wait_for_username` variable to `true` in the constructor of the `PasswordFormFillData` object. Note that this does not protect against the clickjacking attacks described in Section 4.4 but can be extended to do so.

Minimizing user inconvenience. As always forcing user interaction before autofilling may cause inconvenience to the user, password managers could provide a "autofill-and-submit" functionality that once triggered by user interaction will autofill the login form and submit it. We found that variants of autofill-and-submit are already supported by 1Password, LastPass, Norton IdentitySafe, and KeePass.

With this feature, the user's total interaction will remain similar to the current manual autofill password managers. Instead of interacting with the submit button after the password managers autofill the login form, the user will interact with the password manager to trigger autofill-and-submit. As long as the conditions stated earlier in this section are satisfied, the use of such a feature will be as secure as manually entering a password.

5.2 Secure Filling

Our main defense, called *secure filling*, is intended to make the use of password managers more secure than typing in passwords manually. Simply requiring user interaction is not sufficient. Indeed, if a login page is loaded over HTTP but submitted over HTTPS, no browser or password manager implementation provides security once the login form has been filled with the user's password: JavaScript can read the password directly from the form or change the form's action so that it submits to a password stealing page hosted by the attacker.

The goal of secure filling is that even if an attacker injects malicious JavaScript into the login page, passwords autofilled by the password manager will remain secure so long as the form is submitted over HTTPS. This defense is somewhat akin to `HttpOnly` cookies [10], but applied to autofilled passwords: they can be submitted to the web server, but cannot be accessed by JavaScript. We discuss compatibility issues at the end of the section.

Our proposed defense works as follows:

1. Along with the username and password, the password manager stores the action present in the login form when the username and password were first saved.

2. When a login form is autofilled by the password manager, the password field becomes unreadable by JavaScript. We say that the autofill is now "in progress".

3. If the username or password fields are modified (by the user or by JavaScript) while an autofill is in progress, the autofill aborts. The password is cleared from the password field, and password field becomes readable by JavaScript once more.

4. Once a form with an autofill in progress is submitted, and *after* all JavaScript code that is going to be

[1] Chromium build 231333

run has run, the browser checks that the form's action matches the domain of the action it has stored. If the domains do not match, the password field is erased and the form submission fails. If the domains do match, the form is allowed to submit as normal.

Making the password field unreadable by JavaScript prevents stealth exfiltration, as the malicious JavaScript is unable to read the password field and thus unable to steal the password. Checking the action before allowing the form to submit ensures that the action has not been changed to point to a potentially malicious site. The password is guaranteed to only be filled into a form that submits to the same place as when the password was originally saved. For this to work, it is essential that the check be performed after JavaScript's (and thus the attacker's) last opportunity to modify the form's action.

In the case where the form's action does not match what is stored, it may be desirable to give the user the option to submit the form (and password) anyway. However, the browser should allow the user to make an educated decision by showing the user both the new and original actions and explaining how their password may be leaked. This will weaken security, as the user may chose to submit the form when they should not, but it would improve compatibility when sites undergo a redesign and the login page changes.

Implementation. We implemented a prototype of this defense in Chrome[2] by modifying the `PasswordAutofillAgent` class. In the `FillUserNameAndPassword` method, we fill the password field with a dummy value (a sequence of unprintable characters), then store the real password and the form's action in a `PasswordInfo` object associated with the form. In the `WillSendSubmitEvent` method, we check if the dummy value is still present in the password field; if it is, and if the form's action matches the action we had stored, we replace the dummy value with the real password and allow the form to submit. While our implementation is only a prototype, it shows that implementing this defense is reasonably straightforward, at least in Chrome.

Although browsers vendors will need to implement this functionality in their own password managers, they may consider providing a mechanism for external password manager extensions to implement the same functionality. An API could allow the password manager extension to fill a form and designate it as autofilled, as well as designate the expected action on the form. The behavior would then be the same as with the internal password manager: the password field would become unreadable by JavaScript, and the browser checks that the action has not changed before submitting the form.

[2]Chromium build 231333

5.2.1 Limitations of secure filling

The secure filling approach will cause compatibility issues with existing sites whose login process relies on the ability to read the password field using JavaScript.

AJAX-based login. Some sites submit their login forms using AJAX instead of standard form submission. When the login form's submit button is pressed, these sites use JavaScript to read the form fields, then construct and submit an XMLHttpRequest object. This approach is not compatible with our solution, as JavaScript would not be able to read the filled password field and therefore be unable to construct the XMLHttpRequest. Furthermore, this does not use the form's action field, and therefore the password manager cannot detect when the password is being submitted to a different site than when it was first saved.

To study the impact our proposal would have on existing popular sites, we looked for the use of AJAX for login on the Alexa Top 50 sites, as of October 26, 2013. 10 of the these 50 sites used AJAX to submit logins. 8 of 10 sites were based in China, with only one Chinese site on the list not using AJAX. The remaining two sites were based in Russia and the U.S., with other sites from both countries using ordinary form submission. This suggests the use of AJAX to submit passwords is popular in China but not common elsewhere in the world, and overall AJAX is used by a significant minority of popular sites.

We propose two workarounds that will allow our solution to work with AJAX. First, sites could place the login form in an iFrame instead of using XMLHttpRequest. The iFrame would submit using standard form submission. Using this approach, there is no need for JavaScript to read the form fields and the form's action behaves normally. Therefore, it is fully compatible with our secure filling recommendation, but still allows the user to login asynchronously.

Second, for sites that must use XMLHttpRequest, the browser could provide an additional API that allows JavaScript to submit the password without being able to read it. The existing XMLHttpRequest API uses a send() method to send data. We propose an additional method, sendPassword(). The sendPassword() method accepts a form as a parameter, and sends the contents of the form's password fields without ever making them readable to other JavaScript. To prevent an attacker from exfiltrating a password using AJAX, the password manager should check that whenever a filled password is sent using send-

Password(), the destination URL matches the destination URL from the first time the filled password was sent.

Although these workarounds will require modifications to a few existing sites, the security benefits are well worth the effort. The only downside for sites that do not make the required modifications is that their users will not be able to use some password managers.

Preventing self exfiltration attacks. Chen et al. [17] point out that in some cases an attacker can extract data using what they call "self-exfiltration." In our setting this translates to the following potential attack: if any page on the victim site supports a public discussion forum, an attacker can cause the secure filling mechanism to submit the password to the forum page and have the password posted publicly. The attacker can later visit the public forum and retrieve the posted passwords. Since the attacker is changing the login form's action to another page in the *same domain* our secure filling mechanism will allow the password to be sent. In this discussion, the public forum can be replaced by any public form-posted data on the victim site

For this attack to work, the `name` of the password field on the login page must be the same as the name of the text field on the public forum page. An attacker can easily accomplish this by sending to the browser a login page with the desired name.

Fortunately, it is straight forward to defend against this issue: our secure filling mechanism should only fill a password field whose `name` matches the `name` of the field when the password was saved. Furthermore, dynamically changing the `name` attribute using JavaScript should cause a fill to abort. This defense prevents the attacker from submitting the password using any field with a name`name` other than the one chosen by the site itself for the login page. This prevents the self exfiltration attack, except for the extremely unlikely event where a public forum page on the victim site has a text field whose `name` happens to be identical to the password field `name` on the login page.

User registration pages. An additional limitation of our secure filling proposal is that it cannot improve the security of manually entered passwords. HTML does not provide a way to distinguish between password fields on user registration pages and password fields in login forms. Registration pages frequently use JavaScript to evaluate passwords before submission — for example, to check password strength or to verify two passwords match. Therefore, JavaScript on registration pages must have access to the password.

There are two solutions to this problem. One option is to forbid JavaScript from reading any password field, and require that registration pages use regular text fields programmatically made to behave like password fields. On every key stroke JavaScript on the page replaces the character with an asterisk, as in a password field. To the user the text field will behave as a password field, yet JavaScript on the registration page will be able to access the password.

Alternatively, HTML can be slightly extended to support two types of password fields, one for login and one for registration. For login, the Password field allows no JavaScript access to its contents as needed for *secure fill*. The PasswordRegistration field used for registration allows JavaScript access to its contents but is never autofilled with a saved password (separate password manager features such as a password generator can continue to work).

5.3 Server-side defenses

How can a site defend itself without support from password managers? As the attacks rely on decisions made client-side by the user's password manager, a complete server-side defense is not possible. However, a few existing best-practices can be used to greatly reduce the attack area:

1. Use HTTPS on both the login page and page it submits to. Ideally, use HTTPS everywhere on the site and enable HSTS (HTTP Strict Transport Security) to prevent pages from ever loading under HTTP.

2. Use CSP (Content Security Policy) to prevent the execution of inline scripts, making the injection of JavaScript directly into the login page ineffective.

3. Host the login page in a different subdomain that the rest of the site (i.e., login.site.com instead of site.com). This limits the number of pages considered same-origin with the login page, reducing the attack surface.

None of these defenses are unique to the attacks we described, but are best-practices that will make our attacks more difficult. Even with these defenses, attacks are still possible — attacks that take advantage of broken HTTPS, for example, will still be feasible. Therefore, it remains important that password managers implement the fixes we described to fully defend against the attacks.

6 Related work

There have been several prior works about finding vulnerabilities in existing password managers as well as building stronger password authentication systems. We summarize them below.

Vulnerabilities in password managers: Belekno et al. [11] and Gasti et al. [25] surveyed several password managers and found that most of them save passwords to device storage in an insecure manner. However, these attacks have a very different threat model than the attacks described in this paper. They require the attacker to have physical access to a user's device. By contrast, for our attacks we only consider network attackers which is a weaker threat model than the ones requiring physical access.

Besides autofilling of passwords, several password managers also support autofilling of forms with information like name, phone no etc. Prior works [21, 35, 27] have shown that an attacker can steal autofilled information by using specially crafted forms. This is a different class of attack than the attacks on login forms as unlike login passwords, information filled into these forms is not tied to any particular origin. However, for completeness, we summarize our findings about attacks against autofilling of regular forms in Appendix A.

Some existing works [23, 2] have demonstrated how an attacker can use injected JavaScript to steal user's stored passwords in a password manager for login pages that are either vulnerable to XSS attacks or are fetched over HTTP. However, unlike our attacks, these attacks require that users willingly visit the vulnerable website at the presence of the attacker. Reverse Cross-Site Request (RCSR) [13] vulnerabilities perform phishing attacks by leveraging the fact that several password managers will fill in passwords to login forms even if the form's action differs from the action when the password was first saved. These attacks require that the user clicks the submit button. By contrast, our attacks are completely automated and transparent to the user.

The most closely related works to the attacks we present in this paper are by RSnake [37] and Saltzman et al. [40]. RSnake [37] speculated that an attacker can exploit form autofilling tools that fills forms without any user input in sites vulnerable to XSS attacks to extract the autofillable information without users' notice. The basic idea is to inject JavaScript using the XSS attack and exfiltrate the autofilled information. Saltzman et al. [40] suggested that active network attackers can inject iFrames to login forms of websites vulnerable to script injection either through XSS attacks or through pages loaded over HTTP, make the password managers fill those login forms, and steal those passwords without users noticing anything wrong. However, none of these works tested the attacks. We performed a comprehensive study of vulnerabilities and presented several new and different attack vectors (mixed content, broken SSL, embedded device admin pages etc.) and attack techniques (such as the redirect attack).

Using XSS attacks for stealing autofilled passwords has also been explored by Stock et al. [42]. They suggested that the password managers can prevent such attacks by using a placeholder dummy password for autofilling and replacing it with the original one just before submitting the login form to the remote server. In this work, unlike Stock at al., we explore several different vectors for stealing autofilled passwords besides XSS attacks. We also investigate several different third-party password managers together with the builtin password managers that were analyzed by Stock et al.

Blanchou et al. [12] describe several weaknesses of password manager browser extensions and implement a phishing attack that demonstrates the danger of automatic autofill. They do not examine any built-in browser password managers or consider how passwords from multiple sites could be stolen in one attack. They suggest that password managers prevent the cross-domain submission of passwords (what we called action exfiltration in this paper), but do not consider stealth exfiltration.

Fahl et al. [22] demonstrate attacks against Android password managers. However, their attacks were specific to the Android operating system, and most relied upon a malicious Android app, not a network attacker.

Li et al. [32] survey a variety of vulnerabilities specific to third-party web-based password managers and a web attacker, then discuss mitigation strategies. They do not discuss browser or native code password managers, nor a network attacker.

Both the Chromium and Firefox bug databases have bugs filed to prevent autofilling of login information inside an iFrame [18, 16]. However, preventing autofilling of passwords inside iFrames will not prevent the window sweep or the redirect attacks described in Section 4. At the time of this writing, only the Chromium bug has been fixed.

Another Chromium bug [19] seeks to only autofill forms after the user interacts with the login page, but not necessarily the login form. This is not yet implemented, however, increasing the scope of interaction to the entire page will make it easier for the attackers to launch clickjacking attacks. In contrast, autofilling only after explicit user interaction with the login form as suggested in Section 5 is robust against such attacks.

A Firefox bug [14] discusses man-in-the-middle attacks against the password manager similar to our redirect attack. Another bug [15] suggests that filled passwords should not be readable by JavaScript. Their approach is similar to our secure filling, but remains vulner-

able to exfiltration using the action attribute. Although both bugs are several years old, neither has been acted upon.

Password manager features: Aris [9] discusses the autocomplete attribute and why setting autocomplete=off results in poor security in addition to a bad user experience.

Secure password authentication systems: Another related line of research investigated designing secure password authentication systems that can choose strong domain-specific passwords with minimal user intervention [36, 28]. The main motivation behind these works is to minimize the damage caused by users mistakenly revealing their passwords through phishing websites or social engineering These solutions also protect against an attacker leveraging reused passwords that were stolen from a low security website on a high security website. None of these works focus on autofilling of passwords and thus do not help in preventing against the attacks we presented in this paper.

There are also several research works that built password authentication systems that supported autofilling [45, 44]. However, their primary goal was to prevent phishing attacks. In this paper, we focus on existing password managers and thus do not evaluate how vulnerable these systems are against our attacks.

Sandler et al. proposed the 'password booth', a new secure browser-controlled mechanism to let users securely enter passwords that are not unaccessible from JavaScript running as part of the host page's origin [41]. Their solution is similar to our secure filling defense, but does not take password managers into account. Secure filling takes advantage of password managers to provide guarantees the password booth cannot, namely that an autofilled password is submitted to the same origin it was saved from. Furthermore, their proposal requires a dramatic UI change for all users, whereas ours requires only a very minimal UI change from automatic to manual autofill. They suggest that a dramatic change is a feature because it makes security more visible to the user, yet at the same time a dramatic change will reduce adoption from browser developers unwilling to upset their users with change. Ultimately, our two ideas are compatible as the password booth could be extended to work with password managers as we describe in this paper.

An early unpublished version of this paper, containing only a subset of the results, appears as a technical report in [34].

7 Conclusions

In this paper we surveyed a wide variety of password managers and found that they follow very different and inconsistent autofill policies. We showed how an evil coffee shop attacker can leverage these policies to steal the user's stored passwords without any user interaction. We also demonstrated that password managers can prevent these attacks by simply following two steps - never autofilling under certain conditions like in the presence of HTTPS certificate validation errors and requiring user interaction through some form of trusted browser UI, that untrusted JavaScript cannot affect, before autofilling any passwords. Finally, we presented *secure filling*, a defense that makes autofilling password managers more secure than manually entering a password under certain circumstances (e.g., a login page fetched over HTTP but submitted over HTTPS). We hope that this work will improve the security of password managers and encourage developers to adopt our enhancements.

We disclosed our results to the password manager vendors, prompting several changes to autofill policies. Due to our findings, LastPass will no longer automatically autofill password fields in iFrames, and 1Password will no longer offer to fill passwords from HTTPS pages on HTTP pages.

Acknowledgments

This work was supported by NSF, the DARPA SAFER program, and a Google PhD Fellowship to Suman Jana. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, or Google.

References

[1] 1password - agilebits. `https://agilebits.com/onepassword`.

[2] Abusing password managers with xss. `http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/`.

[3] The autocomplete attribute. `http://www.w3.org/TR/2011/WD-html5-20110525/common-input-element-attributes.html#the-autocomplete-attribute`.

[4] Keepass password safe. `http://keypass.info`.

[5] Lastpass — the last password you have to remember. `https://lastpass.com`.

[6] Norton identity safe: Password manager & online identity security. `https://identitysafe.norton.com`.

[7] Secure password manager - keeper password & data vault1password. `https://keepersecurity.com`.

[8] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX Security Symposium*, 2013.

[9] Aris. The war against autocomplete=off, 2013. http://blog.0xbadc0de.be/archives/124.

[10] A. Barth. Http state management mechanism. RFC 2965, 2011.

[11] A. Belenko and D. Sklyarov. secure password managers and military-grade encryption on smartphones: Oh, really? *Blackhat Europe*, 2012.

[12] M. Blanchou and P. Youn. Password managers: Exposing passwords everywhere, 2013. https://isecpartners.github.io/whitepapers/passwords/2013/11/05/Browser-Extension-Password-Managers.html.

[13] Bugzilla@Mozilla. Bug 360493 - (cve-2006-6077) cross-site forms + password manager = security failure. `https://bugzilla.mozilla.org/show_bug.cgi?id=360493`.

[14] Bugzilla@Mozilla. Bug 534541 - passwords from login manager can be intercepted by mitm attacker (e.g. evil wifi hotspot or dns poisoning). `https://bugzilla.mozilla.org/show_bug.cgi?id=534541`.

[15] Bugzilla@Mozilla. Bug 653132 - auto-filled password fields should not have their values available to javascript). `https://bugzilla.mozilla.org/show_bug.cgi?id=653132`.

[16] Bugzilla@Mozilla. Bug 786276 - don't autofill passwords in frames that are not same-origin with top-level page. `https://bugzilla.mozilla.org/show_bug.cgi?id=786276`.

[17] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *W2SP*, 2012.

[18] Chromium. Issue 163072: Chrome should only fill in saved passwords after user action. `https://code.google.com/p/chromium/issues/detail?id=163072`.

[19] Chromium. Issue 257156: Don't autofill passwords on page load for iframed content. `https://code.google.com/p/chromium/issues/detail?id=257156`.

[20] S. Corp. 2013 internet security threat report, volume 18. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf`.

[21] J. de Valk. Why you should not use autocomplete. `https://yoast.com/autocomplete-security/`.

[22] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, you, get off of my clipboard. In *Financial Cryptography and Data Security*, pages 144–161. Springer, 2013.

[23] M. Felker. Password management concerns with ie and firefox, part one, 2010. `http://www.symantec.com/connect/articles/password-management-concerns-ie-and-firefox-part-one`.

[24] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. Xss exploits: Cross site scripting attacks and defense. *Syngress*, 2(3), 2007.

[25] P. Gasti and K. B. Rasmussen. On the security of password manager database formats. In *ESORICS*. 2012.

[26] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward secure embedded web interfaces. In *USENIX Security Symposium*, 2011.

[27] J. Grossman. I know who your name, where you work, and live (safari v4 & v5). `http://jeremiahgrossman.blogspot.com/2010/07/i-know-who-your-name-where-you-work-and.html`.

[28] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *WWW*, 2005.

[29] R. Hansen. Clickjacking. `http://ha.ckers.org/blog/20080915/clickjacking/`.

[30] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). `http://www.hjp.at/doc/rfc/rfc6797.html`.

[31] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: attacks and defenses. In *USENIX Security Symposium*, 2012.

[32] Z. Li, W. He, D. Akhawe, and D. Song. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)*, Aug. 2014.

[33] M. Marlinspike. New tricks for defeating ssl in practice. In *Blackhat DC*, 2009.

[34] R. Gonzalez, E. Chen, and C. Jackson. Automated password extraction attack on modern password managers. Unpublished, Sep. 2013. `arxiv.org/pdf/1309.1416v1.pdf`.

[35] R. M. Rodriguez. How to take advantage of chrome autofill feature to get sensitive information. `http://blog.elevenpaths.com/2013/10/how-to-take-advantage-of-chrome.html`.

[36] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. Stronger password authentication using browser extensions. In *Usenix Security Symposium*, 2005.

[37] RSnake. Stealing user information via automatic form filling. `http://ha.ckers.org/blog/20060821/stealing-user-information-via-automatic-form-filling`.

[38] RunSSL. Ssl certificate for private internal ip address or local intranet server name. `http://runssl.com/members/knowledgebase/9/SSL-Certificate-For-Private-Internal-IP-Address-or-Local-Intranet-Server-Name.html`.

[39] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP*, 2010.

[40] R. Saltzman and A. Sharabani. Active man in the middle attacks. *OWASP AU*, 2009.

[41] D. Sandler and D. S. Wallach. input type=password must die. *W2SP*, pages 102–113, 2008.

[42] B. Stock and M. Johns. Protecting Users Against XSS-based Password Manager Abuse. In *AsiaCCS*, 2014.

[43] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, 2009.

[44] M. Wu, R. C. Miller, and G. Little. Web wallet: preventing phishing attacks by revealing user intentions. In *SOUPS*, 2006.

[45] K.-P. Yee and K. Sitaker. Passpet: convenient password management and phishing protection. In *SOUPS*, 2006.

A Autofilling of forms

Several password managers (Chrome, Safari, LastPass and 1Password) that we studied in this paper also supported autofilling forms with different pieces of information like name, email address, phone no, credit card no, expiry date etc. Even though this is not directly related to autofilling of passwords we summarize our findings in this section for completeness.

Unlike login information, autofill information for forms is not tied to any origin. Therefore, forms from any domain can be autofilled with the same information. To make autofilling secure all the password managers we studied required user interaction to start autofilling of forms. However, several prior works have noticed that a malicious attacker can create specially crafted forms that only have certain innocuous fields visible (e.g. name) while other more sensitive fields (e.g. phone number) invisible to the user and once the user triggers autofilling, both the invisible and visible fields get filled and thus become accessible by the attacker [21, 35].

We found that while all the autofilling password managers we studied are to some extent vulnerable to this attack, the type of sensitive information that can be extracted depends on the nature of user interaction required to trigger autofill. Unlike the rest of the paper in this section we consider web attackers only as the autofill information is not tied by any origin.

- **Chrome & Safari:** Both Chrome and Safari separate the autofillable information into two categories - personal information (e.g., name, email address, phone no., physical address) and credit card information (e.g., credit card no, expiry date). To trigger autofill for each category the user needs to click a field in each category and select an entry from the available ones. Thus, even if an attacker makes a user click a visible field in the personal information category none of the hidden credit card fields will get autofilled. This makes stealing credit information much harder in these password managers without the users noticing it.

- **LastPass:** Unlike Chrome and Safari, for triggering autofilling, LastPass only requires that user click a button shown on top of the page. Once this button is clicked all fields in the form (both hidden and visible) gets filled. This makes it very easy for an attacker to create a crafted form showing only fields like name and email address while stealing additional information, such as credit cards, or a Social Security Number, through hidden fields.

- **1Password:** Unlike LastPass, 1Password requires that the users click different buttons depending on what information they want to fill. Thus, it is not possible to steal credit card information from 1Password by making all credit cards hidden. However, if a legitimate page that a user wants to fill credit card information into also contains an iFrame with hidden credit card fields from a third-party domain (e.g., advertisement), 1Password will fill the credit card information inside the iFrame as well as in the main page with a single click and no notification to the user.

The Emperor's New Password Manager: Security Analysis of Web-based Password Managers

Zhiwei Li, Warren He, Devdatta Akhawe, Dawn Song
University of California, Berkeley

Abstract

We conduct a security analysis of five popular web-based password managers. Unlike "local" password managers, web-based password managers run in the browser. We identify four key security concerns for web-based password managers and, for each, identify representative vulnerabilities through our case studies. Our attacks are severe: in four out of the five password managers we studied, an attacker can learn a user's credentials for arbitrary websites. We find vulnerabilities in diverse features like one-time passwords, bookmarklets, and shared passwords. The root-causes of the vulnerabilities are also diverse: ranging from logic and authorization mistakes to misunderstandings about the web security model, in addition to the typical vulnerabilities like CSRF and XSS. Our study suggests that it remains to be a challenge for the password managers to be secure. To guide future development of password managers, we provide guidance for password managers. Given the diversity of vulnerabilities we identified, we advocate a defense-in-depth approach to ensure security of password managers.

1 Introduction

It is a truth universally acknowledged, that password-based authentication on the web is insecure. One primary, if not *the* primary, concern with password authentication is the cognitive burden of choosing secure, random passwords across all the sites that rely on password authentication. A large body of evidence suggests users have—possibly, rationally [20]—given up, choosing simple passwords and reusing them across sites.

Password managers aim to provide a way out of this dire scenario. A secure password manager could automatically generate and fill-in passwords on websites, freeing users from the cognitive burden of remembering them. Additionally, since password managers automatically fill in passwords based on the current location of the page, they also provide some protection against phishing attacks. Add cloud-based synchronization across devices, and password managers promise tremendous security and usability benefits at minimal deployability costs [10].

Given these advantages, the popular media often extols the security advantages of modern password managers (e.g., CNET [11], PC Magazine [29], and New York Times [32]). Even technical publications, from books [12, 34] to papers [19], recommend password managers. A recent US-CERT publication [21] notes:

> [A Password Manager] is one of the best ways to keep track of each unique password or passphrase that you have created for your various online accounts without writing them down on a piece of paper and risking that others will see them.

Unsurprisingly, users are increasingly looking towards password managers for relieving password fatigue. LastPass, a web-based password manager that syncs across devices, claimed to have over a million users in January 2011 [25]. PasswordBox, launched in May 2013, claims to have over a million users in less than three months [42].

Our work aims to evaluate the security of popular password managers in *practice*. While idealized password managers provide a lot of advantages, implementation flaws can negate all the advantages of an idealized password manager, similar to previous results with other password replacement schemes such as SSOs [40, 38]. We aim to understand the current state of password managers and identify best practices and anti-patterns to guide the design of current and future password managers.

Widespread adoption of insecure password managers could make things worse: adding a new, untested single point of failure to the web authentication ecosystem. After all, a vulnerability in a password manager could allow an attacker to steal *all* passwords for a user in a single swoop. Given the increasing popularity of pass-

word managers, the possibility of vulnerable password managers is disconcerting and motivates our work.

We conduct a comprehensive security analysis of five popular, modern web-based password managers. We identified four key concerns for modern web-based password managers: bookmarklet vulnerabilities, "classic" web vulnerabilities, logic vulnerabilities, and UI vulnerabilities. Using this framework for our analysis, we studied each password application and found multiple vulnerabilities of each of the four types.

Our attacks are severe: in four out of the five password managers we studied, an attacker can learn a user's credentials for arbitrary websites. We find vulnerabilities in diverse features like one-time passwords, bookmarklets, and shared passwords. The root-causes of the vulnerabilities are also diverse: ranging from logic and authorization mistakes to misunderstandings about the web security model, in addition to vulnerabilities like CSRF and XSS.

All the password manager applications we studied are proprietary and rely on code obfuscation/minification techniques. In the absence of standard, cross-platform mechanisms, the password managers we study implement features like auto-fill, client-side encryption, and one-time password in diverse ways. The password managers we study also lack a published security architecture. All these issues combine to make analysis difficult.

Our main contribution is systematically identifying the attack surface, security goals, and vulnerabilities in popular password managers. Modern web-based password managers are complex applications and our systematic approach enables a comprehensive security analysis (in contrast to typical manual approaches).

Millions of users trust these vulnerable password managers to securely store their secrets. Our study strikes a note of caution: while in theory password managers provide a number of advantages, it appears that real-world password managers are often insecure.

Finally, to guide future development of password managers, we provide guidance for password managers. We identify anti-patterns that could hide more vulnerabilities; architectural and protocol changes that would fix the vulnerabilities; as well as identify mitigations (such as Content Security Policy [14]) that could have mitigated some vulnerabilities. Our focus is not on finding fixes for the vulnerabilities we identified; instead, our guidance is broader and aims to reduce and mitigate any future vulnerabilities. Given the diversity of vulnerabilities we identified, we believe a defense-in-depth approach has the best shot at ensuring the security of password managers.

Ethics and Responsible Disclosure. We experimentally verified all our attacks in an ethical manner. We reported all the attacks discussed below to the software vendors affected in the last week of August 2013. Four out of the five vendors responded within a week of our report, while one (NeedMyPassword) still has not responded to our report. Aside from linkability vulnerabilities and those found in NeedMyPassword, all other bugs that we describe in the paper have been fixed by vendors within days after disclosure. None of the password managers had a bug bounty program.

Alice	a legitimate user
Bob	a legitimate collaborator
hunter2	an example password
`dropbox.com`	a benign web application
`facebook.com`	a benign web application
`/login`	entry point (login page) for a web application
Mallory	an attacker
Eve	an attacker
`evil.com`	a website controlled by an attacker
`dropbox.com`	The dropbox.com JavaScript code running in the browser

Figure 1: Naming convention used in the paper. URLs default to https unless otherwise specified.

Organization. We organize the rest of the paper as follows: Section 2 provides background on modern web-based password managers and their features. We also articulate their security goals and explain our threat model in Section 2. Next, we present the four key sources of vulnerabilities we used to guide our analysis (Section 3). Section 4 presents our study of five representative password managers, broken down by the source of vulnerabilities (per Section 3). We provide guidance to password managers in Section 5. We present related work in Section 6 before concluding (Section 7).

2 Background

To start, we explain the concept of a password manager and discuss some salient features in modern implementations. We also briefly list the password managers we studied, identify the threat model we work with, and the security goals for web-based password managers. Here and throughout this paper, we rely on a familiar naming convention (presented in Figure 1) to identify users, web applications, and attackers.

2.1 A Basic Password Manager

At its core, a password manager exists as a database to store a user's passwords and usernames on different sites. The password manager controls access to this database via a *master username/password*. A secure password manager, with a strong master password, ensures that a user can rely on distinct, unguessable passwords for each web application without the associated cognitive burden of memorizing all them. Instead, the user only has to

remember one strong master password.

A password manager maintains a database of a user's *credentials* on different *web applications*. A web application is a site that authenticates its users by asking for a username/password combination. The web application's "entry point" is the page where the application's user can enter her username and password. We call the combination of an entry point, username, and password a *credential*. A user can store multiple credentials for the same web application, in which case a name distinguishes each (typically the username).

Figure 2 (a) illustrates the general protocol of how a user (Alice) uses a password manager (e.g., LastPass) to log in to a web application (e.g., Dropbox). Alice first logs in to the password manager using her master username/password (her LastPass username and password), as shown in Step ①. Then, in Step ②, Alice retrieves her credential for `dropbox.com`. Finally, Alice uses this credential to log into `dropbox.com` in Step ③ and ④.

Since manually retrieving and sending credentials is cumbersome, password managers may also automate the process of selecting the appropriate credential and logging in to the opened web application. This may include navigating a web browser to the entry point, filling in some text boxes with the username/password, and submitting the login form. Since these tasks involve executing code inside the web application, password managers often rely on a privileged browser extension or a bookmarklet for the same.

2.2 Features in Modern Password Managers

Modern password managers provide a number of convenience and security features that are relevant to a security analysis. We briefly elucidate three below.

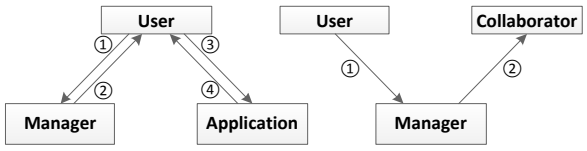


Figure 2: Different parties in a password manager scheme

Collaboration. Modern password managers include the ability to share passwords with a collaborator. Figure 2 (b) illustrates the general protocol of how a user Alice shares a credential of hers with a collaborator Bob. In Step ①, Alice requests that the password manager share a specified credential with Bob. In Step ②, the password manager forwards the credential to Bob when Bob requests it. Both Alice and Bob need accounts with the password manager. My1login even allows the password owner to set read/write permissions on the shared credentials, but the efficacy of these fine-grained controls is not clear, since denying write access does not prevent a collaborator from going to the web application and changing the account's password.

Credential Encryption. Due to the particularly sensitive nature of the data handled by password managers, password managers aim to minimize the amount of code and personnel with access to the credentials in the clear. One common technique is encrypting the credential database on the user's computer, thus preventing a passive attacker at the server-side from accessing the credentials in plaintext. In web-based password managers, this corresponds to using JavaScript to encrypt passwords on the client side (including pages on the password manager's website, browser extensions, and bookmarklets). The password manager encrypts/decrypts the credential database using a key derivation function starting from a user provided secret. If the password manager supports credential encryption, we call the encryption key the user's *master key*. For example, LastPass uses JavaScript to decrypt/encrypt the user's credential database using a key derived from the user's master username and password.

Login Bookmarklets. As discussed above, password managers typically rely on browser extensions to implement auto-fill and auto-login functionality. Unfortunately, users can only install these in a browser that supports extensions. With the popularity of mobile devices whose browsers lack support for extension APIs (e.g., Mobile Safari or Internet Explorer), password managers have adopted a more portable solution by providing a *bookmarklet*. A bookmarklet is a snippet of JavaScript code that installs as a bookmark, which, instead of navigating to a URL when activated, runs the JavaScript snippet in the (possibly malicious) context of the current page (e.g., `evil.com`). This allows the password manager to interact with a login form using widely supported bookmarking mechanisms.

2.3 Representative Password Manager Applications

To evaluate the security of modern password managers, we studied a representative sample of five modern password managers supporting a diverse mix of features. Table 1 provides an overview of their features. The columns "Extension" and "Bookmarklet" indicate support for login automation through the particular mechanism; "Website" indicates the presence of a web-based account management interface; and "Credential Encryption" and "Collaboration" refer to the features described in Section 2.2. For password managers supporting credential encryption, Table 1 also lists their key derivation

	Bookmarklet	Extension	Website	Credential Encryption		Collaboration
				Master Key Derivation	Encrypted Fields	
LastPass	✓	✓	✓	KDF(mp,mu,5000,32)	usernames and passwords	✓
RoboForm	✓	✓	✓	×		×
My1login	✓	×	✓	MD5(ph_{even})+MD5(ph_{odd})	usernames and passwords	✓
PasswordBox	×	✓	×	KDF(mp,mu,10000,32)	passwords only	✓
NeedMyPassword	×	×	✓	×		×

mu: master username mp: master password
ph: passphrase $ph_{even(odd)}$: characters at even (odd) positions of ph
KDF(p,s,c,l) is a key derivation function [23], which derives key of length l octets for the password p, the salt s, and the iteration count c.

Table 1: List of Password Managers Studied.

function and the fields encrypted.

2.3.1 LastPass

LastPass [24] is a popular, award-winning password manager available on phones, tablets, and desktops for all the major operating systems and browsers. It is the top-rated and Editors' Choice password manager for both PC Magazine [29] and CNET [11]. As of August 2013, LastPass had over one million users.

LastPass is one of the most full-featured password manager applications available. It supports nearly all major browsers and mobile/desktop platforms and includes features such as bookmarklets, one-time passwords, and two-factor authentication. LastPass users can access their credentials using the LastPass extension, through a bookmarklet, or directly through the LastPass website. LastPass stores the credential database encrypted on the LastPass servers and also allows users to share passwords with each other.

2.3.2 RoboForm

RoboForm (Everywhere) [33] is another top-rated password manager [29].[1] In RoboForm, each credential (i.e., username, password, and entry point tuple) has its own file named (by default) after the web application's domain. For example, RoboForm uses "dropbox" as the default filename when saving credentials for `dropbox.com`. The user can also choose arbitrary names for the files. Unless the user creates a master password to protect the files, these credential files are sent to RoboForm servers in the clear. The user can access her credential files directly through the RoboForm website or via the RoboForm extension or bookmarklet.

[1] RoboForm (Desktop) is a version of RoboForm that only stores credentials on a single computer and does not sync across devices using a web server. We focus only on the web-based RoboForm (Everywhere) software.

2.3.3 My1login

My1login is a web-based password manager, launched in April 2012; it started a special business-targeted product launched in May 2013. Our study was based on a then-beta version of their consumer-facing service. For maximum compatibility, My1login relies exclusively on bookmarklets and does not provide any browser extensions. Users can access credentials via a web application. My1login also supports sharing of credentials between two My1login accounts. My1login stores all credentials encrypted at the server-side with a special passphrase that the user sets up. In contrast to other password managers, which use the standard PBKDF algorithm, My1login concatenates the MD5 hash of odd and even characters of the passphrase to generate a 256-bit key. We do not comment on this further because we found a simpler, more severe flaw in My1login [27].

2.3.4 PasswordBox

PasswordBox [31], a web-based password manager that launched in 2013, is highly rated by both PC Magazine [29] and CNET [11]. Within three months of its inception in May 2013, PasswordBox had attracted over one million users [42]. PasswordBox, unlike other password managers discussed earlier, does not support bookmarklets; instead, it requires users to install a browser extension. PasswordBox also allows sharing credentials between users and encrypts all passwords using a 256-bit key derived using 10000 iterations of PBKDF2 and the PasswordBox username as the salt.

2.3.5 NeedMyPassword

Finally, we also studied a basic password manager named NeedMyPassword [30]. NeedMyPassword lacks common features such as auto-login, credential sharing, and password generation. Instead, it provides only credential storage, accessible through the NeedMyPassword website. User credentials are not encrypted before send-

ing to NeedMyPassword servers.

2.4 Threat Model

Our main threat model is the *web attacker* [2]. Briefly, a web attacker controls one or more web servers and DNS domains and can get a victim to visit domains controlled by the attacker. We believe this is the key threat model for web-based password managers that often run in the browser. For our study, we extend this model a bit: the user may create an account on the attacker's web application and use the password manager for managing the credentials for the same. Our threat model allows the victim to rely on the password manager's extension, the bookmarklet, and website as she sees fit. The attacker can also create accounts in the password manager service and make requests to the password manager directly.

The password manager's code often runs in a web application's origin (via an extension or a bookmarklet). We assume that the password manager's code is not malicious and does not steal sensitive data from web applications. We also assume that the password manager does not share Alice's credentials with user Bob, unless asked to do so by Alice. Additionally, we assume that the user uses a unique password for the password manager and does not share it with other applications such as `evil.com`.

2.5 Security Goal

At a high level, a password manager only has one key security invariant: ensure that a stored password is accessed only by the authorized user(s) and the website the password is for. We discuss how password managers (attempt to) achieve this invariant by following four security goals. A related taxonomy appears in Bonneau et al.'s analysis of general web authentication schemes [10], but ours is a bit different since we focus exclusively on web-based password managers. Nonetheless, all our goals map to goals mentioned in Bonneau et al.'s work. As we present in Section 4, we found attacks that violate *all* of the security goals identified below and range from complete (password manager) account takeover to privacy violations.

Master Account Security. The first goal of password manager application is the integrity of the master account. It should be impossible for an attacker to authenticate as the user to the password manager. It is crucial that the password manager maintain the security of the master account and safeguard credentials such as master password and cookies. In case of password managers that encrypt credentials, the master key/password used to encrypt the credential database should always remain at the client-side.

Credential Database Security. The main responsibility of a password manager is securely storing the list of a user's credentials. A password manager needs to ensure the security—including confidentiality, integrity, and availability—of the credential database. The attacker, Eve, should not be able to learn Alice's credentials, which would allow Eve to log in as Alice; or modify credentials, which would allow Eve to carry out a form of login CSRF attacks; or delete credentials, which would allow Eve to carry out a denial-of-service attack on Alice.

Collaborator Integrity. The collaboration, or sharing, feature in modern password managers complicates credential databases. Now, each credential has an access-control list identifying the list of users allowed to read/write the credential. A password manager must ensure the security of this feature: e.g., flaws in this feature could allow an attacker to learn a user's credential. While we realize that these goals are a subset of the broader goal of credential database security (above), we separated them out to highlight the security concerns of the sharing credentials feature.

Unlinkability. The use of a password manager should not allow colluding web applications to track a single user across websites, possibly due to leaked identifiers. We use the Bonneau et al.'s definition of unlinkability [10]: a password manager violates unlinkability if it allows tracking a user across web applications even in the absence of other techniques like web fingerprinting [16]. For example, a privacy-minded user could rely on different browsers or computers to foil web browser fingerprinting; a password manager should not add a reliable fingerprinting mechanism that makes that effort moot. Such a fingerprinting mechanism would violate the user's privacy expectations. Equivalently, relying on a password manager should not allow a web application to link two accounts owned by the user with the (same) web application.

3 Attack Surface

The key difference between web-based password managers and "local" password managers is their need to work in web browsers. Web-based password managers store credentials in the cloud and a user logs on to the manager to retrieve his/her credentials. Access to the stored credentials is via extensions, a website, or even bookmarklets—all of which run in the browser.

To guide our investigation, we identified four key concerns for modern web-based password managers: bookmarklet vulnerabilities, classic web vulnerabilities, authorization vulnerabilities, and UI vulnerabilities. We discuss each in turn below. In the next section, we will present representative vulnerabilities of each type.

3.1 Bookmarklet Vulnerabilities

JavaScript is a dynamic, extensible language with deep support for meta-programming. The bookmarklet code, running in the context of the attacker's JavaScript context cannot trust *any* of the APIs available to typical web applications—an attacker could have replaced them with malicious code. Relying too much on these APIs has created a class of vulnerabilities unique to web-based password managers.

To fill in a password on (say) `dropbox.com`, a password manager needs to successfully authenticate a user, download the (possibly encrypted) credential, decrypt it (if necessary), authenticate the web application, and, finally, perform the login. Doing all this in an untrusted website's scripting environment (as a bookmarklet does) is tricky. In fact, three of the five password managers we studied (Table 1) provide full-fledged bookmarklet support, and *all* of them were vulnerable to attacks ranging from credential theft to linkability attacks (Section 4).

Browser extensions, which modified the webpage, faced a similar problem in the past. Currently, both Firefox and Chrome instead provide native or isolated APIs for browser extensions. Unfortunately, popular mobile browsers, including Safari on iOS, Chrome on Android/iPhone, and the stock Android Browser, do not support extensions. As a result, web-based password managers often rely on bookmarklets instead.

3.2 Web Vulnerabilities

A password manager runs in a web browser, where it must coexist with the web applications whose passwords it manages as well as other untrusted sites. Unfortunately, relying on the web platform for a security-sensitive application such as password managers is fraught with challenges.

Web-based password manager developers need to understand the security model of the web. For example, browsers share authentication tokens such as cookies across applications (including across applications and extensions), leading to attacks such as cross-site request forgery (CSRF). Applications running in the browser runtime also need to sanitize all untrusted input before inserting it into the document; insufficient sanitization could lead to cross-site scripting attacks, which in the web security model implies a complete compromise.

3.3 Authorization Vulnerabilities

Sharing credentials increases the complexity of securing password managers. While previously, each credential was only accessible by its owner, now each credential needs an access control list. Any user could potentially access a credential belonging to Alice, if Alice has authorized it. A password manager needs to ensure that all actions related to sharing/updating credentials are fully authorized. Confusing authentication for authorization is a classic security vulnerability, one that we find even password managers make (Section 4). We separate out authorization vulnerabilities from web vulnerabilities since they are often due to a missing check at the server-side. For example, all our authorization vulnerabilities involve requests made by an attacker from his own browser, not via Alice's browser (when Alice visits `evil.com`).

3.4 User Interface Vulnerabilities

A major benefit of password managers is their ability to mitigate phishing attacks. Users do not actually memorize the password for a web application; instead, they rely on the password manager to detect which application is open and fill in the right password. The logic that performs this is impervious to phishing attacks: it will only look at the URL to determine which credential to use.

These advantages are moot if the password manager itself is vulnerable to phishing attacks. Even worse, in the case of password managers, a single phishing attack can expose *all* of a user's credentials. Thus, we believe it behooves password managers to take extra precautions against phishing attacks. While it is possible that password managers are susceptible to classic phishing attacks, we focus on anti-patterns that make password managers more vulnerable than the typical website.

For example, consider what happens when a user clicks on a password manager's bookmarklet while not logged in to the password manager. A simple option is asking the user to login in an iframe. Unfortunately, this is trivial for the attacker to intercept and replace the iframe with a fake dialog. Since users cannot see the URL of an iframe, there is no way for a user to identify whether a particular iframe actually belongs to the password manager and is not spoofed. We argue that this is an anti-pattern that password managers should avoid.

4 Security Analysis of Web-based Password Managers

Next, we report the results of our security analysis of five popular password managers. We organize our results per the discussion in Section 3. Table 2 summarizes the vulnerabilities we found. Our discussion below highlights the presence of different types of security vulnerabilities in web-based password managers. We do not present complete architectural details of each password manager; instead, we only provide enough technical details to understand each vulnerability.

4.1 Bookmarklet Vulnerabilities

As discussed earlier, a bookmarklet allows a user of a password manager to log in to web applications without needing to install any extension, a particularly useful

	Bookmarklet Vulnerabilities	Web Vulnerabilities	Authorization Vulnerabilities	User Interface Vulnerabilities
LastPass	✓(§ 4.1.1)	✓(§ 4.2.1)		✓([27])
RoboForm	✓([27])	✓([27])	NA	✓(§ 4.4)
My1login	✓([27])		✓(§ 4.3.1)	
PasswordBox	NA		✓(§ 4.3.2)	NA
NeedMyPassword	NA	✓([27])	NA	NA

Table 2: Summary of Vulnerabilities Discovered. NA identifies vulnerabilities not applicable to the particular password manager because it does not provide the relevant functionality.

feature with mobile browsers that lack extension support. Three of the password managers we studied—LastPass, RoboForm, and My1login—provide access to credentials and auto-fill functionality using bookmarklets. In fact, My1login *only* provides bookmarklet for auto-fill support, advertising it as a feature ("No install needed").

We found critical vulnerabilities in all three bookmarklets we studied. If a user clicks on the bookmarklet on an attacker's site, the attacker, in all three cases, learns credentials for arbitrary websites. We only discuss one representative vulnerability here and provide details of the other two vulnerabilities in our extended technical report [27].

While in 2009 Adida et al. identified attacks on password manager bookmarklets [1], our study indicates that these issues still plague password managers. This is particularly a cause of concern given the popularity of mobile devices that lack full-fledged support for extensions.

4.1.1 Case Study: LastPass Bookmarklet

LastPass stores the credential database on the `lastpass.com` servers encrypted with a `master_key`, which is a 256-bit symmetric key derived from the user's master username and master password. The LastPass client-side code never sends the master password or master key to the LastPass servers.

Recall that a bookmarklet runs in the context of the (possibly malicious) web application. At the same time, due to LastPass's credential encryption, the bookmarklet needs to include the secret `master_key` (or a way to get to it), to decrypt the credential database. Including this secret in the bookmarklet, while still keeping it secret from the web application, is tricky. LastPass also provides the ability to revoke a previously created bookmarklet, further complicating this feature.

Installing a Bookmarklet. A user, Alice, wishing to install a bookmarklet needs to create a special link through her LastPass settings page. On Alice's request, the LastPass page code creates a new random value `_LASTPASS_RAND` and encrypts the `master_key` with it, all within Alice's browser. The LastPass servers then store this encrypted master key (called `key_rand_encrypted`) and an identifier `h` along with

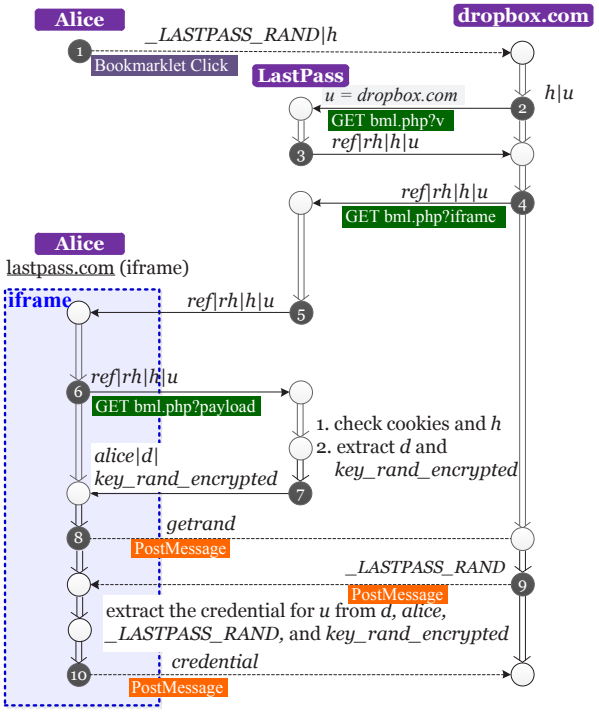


Figure 3: LastPass: Automatic login using bookmarklet. `u` is the domain on which Alice clicked on the bookmarklet.

Alice's credential database. The page then creates a JavaScript snippet containing `_LASTPASS_RAND` and `h`, which Alice can save as a bookmark. This design allows Alice to revoke this bookmarklet in the future by just deleting the corresponding `h` and encrypted master key from the LastPass servers.

Using the Bookmarklet. Figure 3 illustrates how Alice uses her LastPass bookmarklet to log in to `dropbox.com`. At the Dropbox entry point, Alice clicks on her LastPass bookmarklet, which includes the token `_LASTPASS_RAND` and `h`. The bookmarklet code first checks the current page's domain and adds a script element to the page sourced from `lastpass.com`. The request for the script element (Step 2 in Figure 3) sends

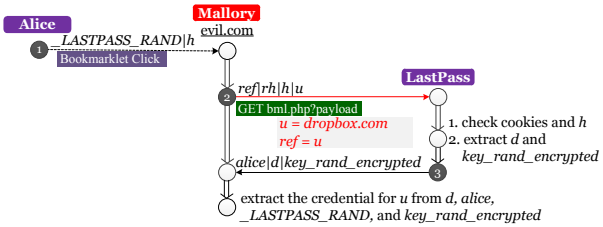


Figure 4: Attack on LastPass bookmarklet based autologin. The `rh,h` values are random; `u` and `ref` identify the Malloy's target website.

`h` and the web application domain `dropbox.com` as parameters `h` and `u`. LastPass checks `h` and if the parameter is valid (i.e., Alice has not revoked the bookmarklet), responds with a JavaScript file containing the additional parameters `ref` and `rh`.

Next, the newly fetched JavaScript file creates an iframe to `lastpass.com` using four parameters: `ref,rh,h,u`. This iframe includes a script located at `lastpass.com/bml.php?u=dropbox.com` that, when downloaded, includes the encrypted master key `key_rand_encrypted` and the credential for `dropbox.com` encrypted with the master key. The iframe then receives the bookmarklet's `_LASTPASS_RAND` value via a `postMessage` call, decrypts the `dropbox.com` credential and sends them back.

Vulnerability. The resource at `bml.php?u=dropbox.com` (Step 6 Figure 3) is at a predictable URI and contains sensitive information. It provides the encrypted master key `key_rand_encrypted` and the credential for `dropbox.com`. The same-origin policy allows an attacker to include a script from any origin and execute it in the attacker's webpage.

LastPass Bookmarklet Attack. Figure 4 illustrates how a malicious web application `evil.com` can steal Alice's credential for `dropbox.com`. When Alice visits the attacker's site `evil.com` and clicks her LastPass bookmarklet, the attacker uses any of a number of hijack techniques [1, 8] (e.g., `Function.toSource`) and extracts both `h` and `_LASTPASS_RAND`. Then, the attacker imitates Step 6 from Figure 3 (as Step 2 here) by writing a `<script>` tag with `src` set to `lastpass.com/bml.php?u=dropbox.com` and adding the parameters `rh` (any string of length 64), `r` (any number), and `h` (from the bookmarklet).

The downloaded script, which runs on the attacker's page, includes all the information needed to decrypt credential for `dropbox.com` (notably, `key_rand_encrypted`). Again, the attacker uses the JavaScript hijack technique to extract out the encrypted credential and decrypts them with the `_LASTPASS_RAND` value stolen earlier. The attacker can repeat the attack to steal *all* of Alice's credentials, violating the confidentiality of the credential database.

LastPass Linkability Attack. Finally, we note that the `h` and `_LASTPASS_RAND` remain the same across browsers but differ by user. As discussed above, any website where the user clicks the bookmarklet can learn these pseudo-identifiers `h` and `_LASTPASS_RAND` [1]. This allows colluding websites to track a user, violating the user's privacy expectations [10]. Additionally, this also allows a single website to identify and link multiple accounts belonging to the same user, which violates the unlinkability goal.

4.2 Web Vulnerabilities

Next, we study vulnerabilities in password managers caused due to subtleties of the web platform. We focus on CSRF and XSS vulnerabilities, which are common in web applications. We find CSRF vulnerabilities in LastPass, RoboForm, and NeedMyPassword as well as XSS vulnerabilities in NeedMyPassword.

Our attacks are severe: XSS vulnerabilities in NeedMyPassword allow for complete account takeover, while the CSRF vulnerabilities in RoboForm allow an attacker to update, delete, and add arbitrary credentials to a user's credential database. We only discuss the CSRF vulnerability in LastPass here and discuss the remaining CSRF and XSS vulnerabilities in our extended technical report[27].

4.2.1 Case Study: LastPass One Time Password

One-Time password (OTP) is a feature of LastPass that allows a user to generate an authentication code for the master account that is only valid for one use. A user can use a one-time password to prevent a physical observer from gaining access to her LastPass account [10].

Generating an OTP. Before getting into the details, we point out that Alice's LastPass OTP must be able to authenticate Alice to LastPass *and* allow Alice to recover her master key; all without revealing anything extra (including the OTP itself) to LastPass servers (since that would defeat the credential encryption feature).

Figure 5 illustrates how Alice creates an OTP `otp`. This starts with Alice creating a string `otp` locally in her browser. Next, Alice computes `h = hash(hash(alice|otp)|otp)` with her LastPass username `alice`. LastPass will use `h` to authenticate Alice, without having to know the exact value of `otp`. Then, Alice encrypts her master key with `hash(alice|otp)`. Alice sends `h` and the encrypted master key (`rand_encrypted_key`) to LastPass. Notice that the LastPass servers never see the generated one-time password or Alice's master key in the clear. LastPass saves a record associating the values `h` and

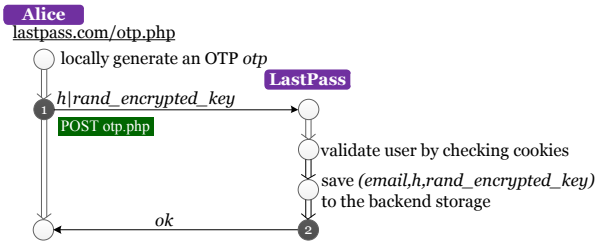


Figure 5: LastPass OTP Creation. Note the absence of any CSRF token in the request in Step 1.

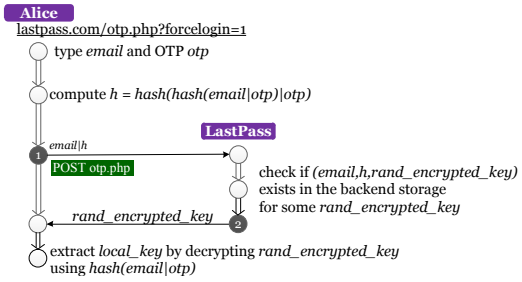


Figure 6: Using the LastPass OTP.`rand_encrypted_key` is the master key encrypted with `hash(alice|otp)`,

`rand_encrypted_key` with Alice's LastPass username.

Using the OTP. To sign in with her OTP (Figure 6), Alice recomputes `h` from her knowledge of `otp`, and sends it to LastPass along with her LastPass username. LastPass checks its records for a matching username and `h`. It starts an authenticated session for (i.e., sets session cookies identifying) Alice and sends back her `rand_encrypted_key`. Alice then decrypts `rand_encrypted_key` to recover her master key.

Vulnerability. We found that the request used to set up the OTP (Step 1 Figure 5) is vulnerable to a classic CSRF attack. The LastPass server authenticates Alice (in Step 1) only with her cookies. Since LastPass does not know the OTP or the master key, it cannot validate that `rand_encrypted_key` actually corresponds to an encrypted value of the master key. Fixing this vulnerability involves adding a CSRF token to the OTP creation form.

OTP Attack on LastPass. An attacker, Mallory, who knows Alice's LastPass username, can come up with a string `otp'` and using the same algorithm as above, generate a forged value `h'` and `rand_fake_key` with a made-up master key. On submitting the CSRF POST request, LastPass will store `h'` as authenticating Alice.

Mallory can then use `otp'` to log-in to LastPass using `otp'`. Of course, decrypting the `rand_fake_key` will not give Mallory Alice's real master key. Nonetheless, using this CSRF attack, Mallory obtains Alice's encrypted password database. We find this leads to three attacks.

First, LastPass stores the list of web application entry points unencrypted, and Mallory can now read this list. This is a breach of privacy: starting with just Alice's LastPass username, Mallory now knows all the web applications Alice has accounts on.

Secondly, the encrypted password database is now available to Mallory for offline guessing. Recall that the LastPass uses a key derived from Alice's *master* password, which Alice has to memorize. Unlike the passwords randomly generated by LastPass, this master password is likely vulnerable to guessing. It is instructive to consider that, after a server breach, LastPass requires all its users to reset their passwords [41].

Finally, we also find that this attack leads to a denial of service attack. Mallory, logged in as Alice, can delete any credential in Alice's database, *despite* being unable to decrypt the database. Since the username is part of the credential, recovering all these credentials would be tedious, or in some cases impossible.

4.3 Authorization Vulnerabilities

Looking beyond vulnerabilities stemming from the nature of the web platform, we now discuss some vulnerabilities that come from logic errors in the password manager. We found that two of the three password managers that support credential sharing both mistake authentication for authorization. An attacker can create two fake accounts, Eve and Mallory, in the password manager and share Alice's credentials with Mallory by sending a correctly crafted message from Eve's account. Importantly, the actual errors do not ever involve Alice or her browser and thus the attacks work in the absence of Alice visiting the attacker's website.

4.3.1 Case Study: My1login Sharing Credentials

My1login relies on client-side encryption of the credential database. This complicates sharing: Alice and Bob need to share credentials, through My1login as an untrusted channel. My1login relies on public-keys for both Alice and Bob to share credentials: when Alice shares a credential with Bob, My1login first encrypts it with Bob's public-key before sending it to Bob. This ensures that only Bob can see the shared credentials.

Sharing My1login Credentials. Figure 7 illustrates how Alice shares a credential with Bob in My1login. In the first two steps, Alice obtains Bob's public key k_b. Then, in Step 3, Alice (i.e., Alice's My1login instance) encrypts the credential with k_b and sends the encrypted username `alice.dropbox@gmail.com` and password `hunter2` to My1login.

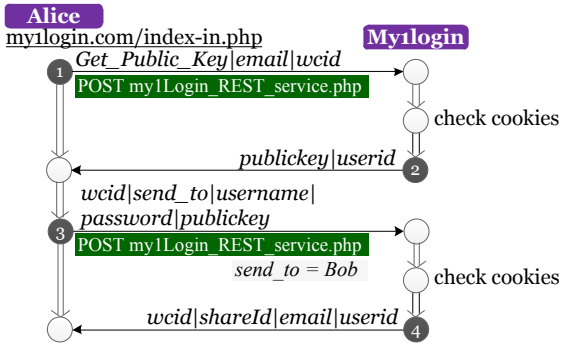


Figure 7: Sharing Credentials on My1login

```
{ "id": 4097211,
  "member_id": 3751238,
  "name": "Dropbox",
  "url": "https :// www.dropbox.com/login",
  "login": "alice .dropbox@gmail.com",
  "note": {},
  "created_at": "2013−07−18T13:50:18−04:00",
  "updated_at": "2013−07−18T13:50:18−04:00",
  "password_k": "AAQsrfjgfcWj/4FsP64BTYTJpbgpBK4+yltal",
  "settings": "{\"autologin\":\"1\", ...} ",
  "member_fullname": "Alice Gordon",
}
```

Listing 1: Example PasswordBox asset

Using the Shared Credential. Bob's My1login instance polls the My1login server for any updates. The My1login server notifies Bob of the newly shared credential, sending him the information that Alice encrypted with his public key. Bob decrypts the shared credentials (`username` and `password`) for website `url` with his private key. Once Alice shares a credential with Bob, he can also update it. In such cases, My1login automatically updates the credential globally by sharing the update with collaborators on the web card (Alice, in this case). This occurs through essentially the same request as Step 3 in Figure 7, but this time Bob encrypts the credential with Alice's public-key.

Vulnerability. Our analysis revealed that My1login only *authenticates* Alice before sharing a web card; it does not check whether Alice owns or has the authority to share the web card identified in the `wcid` (Step 3, Figure 7).

My1login Share Attack. Since My1login does not check `wcid` in Figure 7 Step 3, an attacker Mallory can share any web card (given its id) to a collaborator Eve. This vulnerability allows Mallory to steal any credential whose ID she knows (perhaps because Eve shared it in the past but revoked it later).

Worse, further analysis revealed that web card ids are globally unique, auto-incrementing numbers. In Step 3, Figure 7, Mallory can even use numbers referring to cards not yet created.

Suppose that `wcid` refers to a web card that belongs to (or will belong to) Alice. Mallory generates a dummy username and password like "userabc" and "pwdabcm," encrypts it and shares it with Eve. Eve receives the dummy credentials. While these credentials are useless, notice that this registered Eve as a collaborator on this web card, even if it belongs to Alice.

In the future, whenever Alice or any other collaborator updates the web card, the My1login client automatically re-encrypts the real credential and sends it to each collaborator, *including Eve*. It is trivial for Mallory to share all web cards, current and future, to Eve, who awaits updates to steal real credentials.

In the attack above, Eve learns Alice's credentials only if Alice updates them after the attack. Alternatively, Eve can install new credentials to Alice's database without authorization from Alice. This allows Eve to execute a form of login CSRF attack [5]. Alternatively, Eve can install wrong credentials to Alice's database, which would cause an error when Alice attempts to use them. It is likely that Alice, in response, would update the web card with her correct credentials and unknowingly share them with Eve.

One concern is how to ethically verify the My1login authorization flaw without sharing another user's credential by mistake. We observed over multiple days that it is rare that any other user creates a new web card between 2am - 3am PST. We then verified this vulnerability one day between 2am and 3am without sharing another user's credential by mistake.

4.3.2 Case Study: PasswordBox Sharing Credentials

PasswordBox stores a user's credential for a web application in a JSON-encoded *asset* file. Listing 1 presents an example asset for Dropbox. We focus on two salient properties: first, `password_k` is the encrypted value of Alice's password for `dropbox.com` and is the *only* encrypted field in the asset. Other details such as entry point URL, the name Alice used to register (`member_fullname`) and so on, are all in cleartext.

Second, our analysis revealed that `asset_id` is an auto-incrementing, unique (across all users) id for each asset. Assuming `asset_id` started at 1, we can infer that PasswordBox manages over 4 million assets, an assumption anyone can verify with the flaw we discuss next. (We did not, because of the obvious ethical concerns.)

Sharing Credentials. Figure 8 shows how a user Alice shares one of her assets identified by `asset_id` to a collaborator Bob. On clicking share, the PasswordBox extension on Alice's browser makes a POST request to the `passwordbox.com` servers that includes the

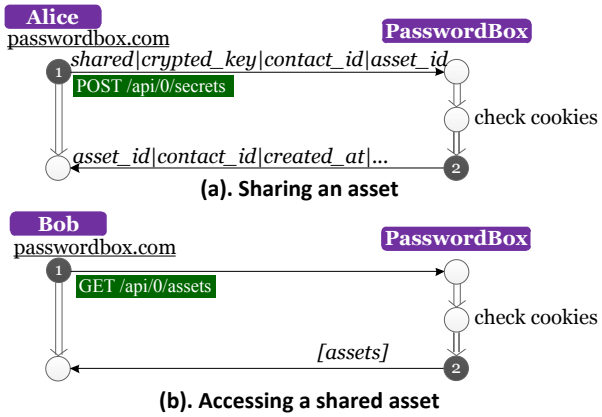


Figure 8: PasswordBox: Sharing an asset. The underlined `passwordbox.com` on the left indicates that the code making the request runs in the `passwordbox.com` origin.

```
function share(asset_id){
  var xmlhttp = new XMLHttpRequest();
  var jsn = '{"shared":true, "crypted_key:" "ABC", "contact_id": 123,
    "asset_id": '  + asset_id + '}';
  xmlhttp.open("POST","https://api0.passwordbox.com/api/0/secrets",true);
  xmlhttp.setRequestHeader("Content−type", "application/json");
  xmlhttp.send(jsn);
}
```

Listing 2: JavaScript snippet to share a asset with Eve

`contact_id`, the contact to share credentials with (in this case, Bob's id); and `asset_id`, the id of the credential to share (as in Listing 1). In the future, whenever Bob downloads the list of assets accessible to him, PasswordBox includes Alice's shared credential.

Vulnerability. The absence of a CSRF token suggested the possibility of a CSRF flaw in the protocol. Fortunately (or, unfortunately), we found that PasswordBox implemented a strong defense against CSRF attacks: it checks the `Referer` header as well as includes a special `X-CSRF-Token` in the headers of the HTTP request. Instead, we found a far more serious logic bug in the sharing assets functionality. In its sharing logic, PasswordBox never checks whether Alice owns the `asset_id` she is sharing. This allows Mallory to share assets she does not own with Eve, similar to the My1login attack (Section 4.3.1).

PasswordBox Share Attack. Similar to the "share-and-update" attack on My1login, Mallory and Eve run through the protocol in Figure 8. Mallory can share any asset to Eve by simply setting `asset_id`. Since `asset_id` is an auto increment number, Mallory can iterate through all possible `asset_id` and share all existing 4 million assets with Eve. Listing 2 is the JavaScript snippet that Mallory used to share an arbitrary asset to Eve, whose `contact_id` is assumed to be `123`.

As we noted above, PasswordBox only encrypts the password field in an asset; disclosure of every user's full name, usernames, web application URLs, and creation times is a severe privacy breach.

4.4 User Interface Vulnerabilities

Earlier, discussing bookmarklet vulnerabilities (Section 4.1), we focused on the behavior of a password manager when the user is already authenticated to the password manager. If the user is not authenticated to the password manager, then the user needs to log in to her master account. This provides a potential avenue for phishing vulnerabilities and the password manager should not train bookmarklet users towards insecure practices. The ideal secure option in such a scenario is asking the user open a new tab (manually) and logging in to the password manager.

We find that only the My1login bookmarklet defaults to this secure behavior. Clicking on the My1login bookmarklet, when not logged in, results in a message asking the user to open a new window and log in. We found that both RoboForm and LastPass bookmarklets were vulnerable to phishing attacks. Below, we discuss the RoboForm vulnerability and present the LastPass vulnerability in our technical report [27]. We also have recorded video demonstrations of these attacks online [4].

Case Study: RoboForm. Recall that when Alice clicks her RoboForm bookmarklet, the bookmarklet creates an iframe in the current web application. If Alice has not logged in to RoboForm, the iframe request redirects to the RoboForm login page, displaying a login form in the iframe. This design is insecure: it trains Alice to fill in her RoboForm password even when the URL bar (belonging to the surrounding web application) does not point to `roboform.com`. An attacker can trivially block the RoboForm iframe load and spoof an authentication dialog that steals Alice's RoboForm credentials. A secure design would ask Alice to open a new tab to RoboForm and log in.

One concern with successfully carrying out this attack is detecting whether Alice is already logged in to RoboForm. We found that the height of the RoboForm iframe (the dialog) is greater than 200px if and only if Alice is already logged-in. Using this side-channel, the attacker can modify the spoofed iframe to make the attack convincing.

5 Lessons and Mitigations

We now attempt to distill the lessons learnt from our study and provide guidance to password managers on closing the vulnerabilities we found and mitigating future ones. Our focus here is on concrete guidance and defense-in-depth. We identify improvements in architectures and protocols to mitigate vulnerabilities as well as the use of browser mitigations like CSP. We also identify anti-patterns that developers of password managers should avoid. Security reviewers and users can also rely on the patterns and (absence of) the mitigations we discuss as indicators of the security of a password manager.

5.1 Bookmarklet Vulnerabilities

All the bookmarklets we studied were vulnerable. The root cause of these vulnerabilities is that the bookmarklet code executes in the untrusted context of the webpage. The web browser guarantees a secure, isolated execution environment for iframes and we advocate an iframe-based architecture for securing password manager bookmarklets. Modern features such as credential encryption, which requires secure client-side code execution, makes the use of defenses proposed in previous work impractical [1].

Recommendation. We recommend password-managers rely on a design similar to proposed by Bhargavan et al. [8]. When the user clicks the bookmarklet, the bookmarklet code loads the password manager code in an iframe, running in the password manager's origin. The browser's same-origin policy isolates code executing in the iframe from the web application page and guarantees integrity of DOM APIs.

The password manager's iframe uses `postMessage` for communicating with the application page and maintains a simple invariant: a message carrying a credential for dropbox.com has a target origin of `https://www.dropbox.com`. The browser guarantees that only the Dropbox page receives the message. The only secret in the bookmarklet code is an HMAC function (protected by DJS [8]) that the password manager iframe can use to provide *click authentication* (i.e., the user actually clicked the bookmarklet). Unfortunately, the presence of the secret in the bookmarklet allows linkability attacks.

For unlinkability, we recommend password managers do not rely on such a secret and HMAC function. Disabling this secret loses the "click authentication" property. Since password manager browser extensions typically include "auto fill" functionality, we believe the loss of click authentication is acceptable. If needed, the code in the password manager iframe could draw a dialog to ask for user confirmation before sharing credentials with the website. Such a design is vulnerable to clickjacking and we also recommend the use of upcoming mitigations for UI security [39].

Instead, password managers could rely on asking the user for permission to share credentials in the iframe created.

The core issue behind bookmarklet vulnerabilities is the absence of secure (or "isolated") DOM APIs for bookmarklets. An alternative possibility is for browser vendors to provide bookmarklets with secure access to these DOM APIs, similar to the access granted to Chrome/Firefox extensions.

5.2 Web Vulnerabilities

We found a number of "classic" web application vulnerabilities in password managers. Based on the critical and sensitive nature of data handled by password managers, we recommend defense-in-depth features such as CSP and identify anti-patterns that developers should beware of.

XSS. XSS is a well-studied problem and we will not recapitulate all the defenses for the same here. We recommend that web applications, in addition to validating input and sanitizing outputs, should also turn on Content Security Policy to provide a second layer of defense against XSS. The absence of a strong CSP policy in a password manager should raise red flags for users and reviewers. In the applications we studied, only LastPass shipped with a Content-Security-Policy header, albeit with an unsafe policy that allows eval and inline scripts.

CSRF. The prevalence of CSRF vulnerabilities in password managers surprised us. We recommend password managers should include CSRF protection (via tokens) for *all* their pages and forms. For defense in depth, these applications should also check the Referer and Origin headers for all requests. While not a reliable defense, these headers provide a useful secondary layer of defense.

One concern with CSRF tokens is the need to create and maintain state at the server-side. This could be cumbersome for password managers that provide an interface through a browser extension: it is infeasible to request a new token before rendering every form. Instead, these applications can rely on special headers (e.g., X-CSRF-Token) for CSRF defense. The web security model disallows `evil.com` from setting headers for a cross-origin request.[2]

Secrets in JavaScript files. An anti-pattern we noticed was the presence of secret values—based off of tokens in the request URI or cookies in the request—in script files. Unfortunately, the web platform does not provide strong isolation guarantees for scripts: any (untrusted) origin can include scripts from the password manager's website. We recommend password managers

[2] Unless explicitly whitelisted by the receiving server via Access-Control-* headers.

serve *all* secret values in HTML or separate JSON files. This requirement is easy to check: the scripts used by the password managers should be the same across all users of the password manager. Serving user-specific JavaScript files based on tokens in the URI is a clear anti-pattern. An alternative is Defensive JavaScript [8], which provides a principled defense to ensure secrecy of values in JavaScript code.

5.3 Authorization Vulnerabilities

The web application vulnerabilities discussed above stemmed from quirks of the web platform (e.g., ambient authentication with cookies). Worryingly, we found a number of *logic* flaws in password managers classified under two broad categories. The first category, insufficient authorization, creates vulnerabilities exacerbated by the second category, predictable identifiers. We identify an anti-pattern, predictable identifiers, and the core security vulnerability, insufficient authorization, below and discuss mitigations.

Insufficient Authorization. Confusing authentication with authorization is a classic security vulnerability. Out of the three password managers that support collaboration, we found insufficient authorization vulnerabilities in two of them. Unfortunately, these are logic flaws, and a simple mitigation is difficult. One possibility is for password managers to use a simpler sharing model. For example, let each credential have only one owner—only the credential's owner can change it or its collaborator list. A simple model eases authorization checks and could make insufficient authorization stand out.

Predictable Identifier. Both our attacks on logic vulnerabilities rely on predictable identifiers (e.g., consecutive integers). We recommend password managers switch to cryptographically secure random numbers for identifiers—this adds defense in depth, even if the server is careful to check authorization. The use of predictable identifiers should be rare and any use should be a cause for a security review. As we discussed earlier, the nature of the data handled by password managers warrants such a default-secure posture.

5.4 User Interface Vulnerabilities

Our proposed solution of relying on iframes and storing tokens in localStorage/cookies works seamlessly only if the user is already logged in. If this is not true, the iframe needs to ask the user to log in. As our attacks demonstrated, the only secure way to do this is asking the user to manually open a new tab and login. My1login is the only password manager relying on this design and we recommend other password managers adopt a similar design. Cautious users can protect themselves against such an attack by always logging in using a new tab instead of trusting a popup or iframe.

6 Related Work

A number of researchers have investigated security of web-based password managers. Bhargavan et al. did a study on five password managers, along with a number of other web services that provide encrypted storage of data in the cloud, and presented a number of web attacks that could violate the intended security of the products [7]. This work inspired a redesign of the LastPass bookmarklet to decrypt a user's credentials inside LastPass's iframe, making it harder for an attacker to steal the master key. Adida et al. provide a comprehensive overview of a number of attacks on password manager bookmarklets; we reuse some of the ideas but find that, with modern password managers relying on encrypted credentials, a new defense based on iframes is needed [1]. Belenko et al. studied the cryptographic properties of password managers for mobile devices and their vulnerability to brute force attacks [6].

In concurrent work, Blanchou and Youn [9] as well as Silver et al. [35] found a number of serious flaws in the auto-fill functionality in password managers. In contrast, we analyze a broader range of functionality but focus on third-party web-based password managers only.

Bonneau et al. [10] presented a framework for evaluating alternatives to passwords in terms of usability, deployability, and security. This framework highlights advantages of an idealized password manager, but our work demonstrates that, in practice, password managers have flaws in their implementations that critically undermine their security. Similarly, recent work found implementation flaws in other password alternatives such as SSOs [40, 38].

The common web attack vectors we considered, such as CSRF and XSS, have seen a lot of work in the past decade. For attacks and defenses, we defer to prior literature for comprehensive surveys on CSRF [43], XSS [18], and server-side defenses for both [26]. Recent work also focused on logic flaws and insufficient authorization in web applications [17, 37, 36].

The security of mutually distrusting JavaScript running in the same origin (an important consideration in bookmarklet code) has not been a concern in the design of the web platform. Bhargavan et al. identified a number of flaws in bookmarklets and proposed a new subset of JavaScript called Defensive JavaScript to mitigate them, which we discussed in depth in Section 5.1. Defensive JavaScript [8] is the only work we are aware of that aims to protect a JavaScript gadget from the host webpage. A large body of work exists for the converse goal of protecting a host webpage from third party JavaScript code (such as code that draws a gadget) [22, 3, 13, 28]; a survey compares these approaches [15].

7 Conclusions

We presented a systematic security analysis of five web-based password managers. We found critical vulnerabilities in all the password managers and in four password managers, an attacker could steal arbitrary credentials from a user's account. Our work is a wake-up call for developers of web-based password managers. The wide spectrum of discovered vulnerabilities, however, makes a single solution unlikely. Instead, we believe developing a secure web-based password manager entails a systematic, defense-in-depth approach. To help such an effort, we provided guidance and mitigations based on our analysis. Since our analysis was manual, it is possible that other vulnerabilities lie undiscovered. Future work includes creating tools to automatically identify such vulnerabilities and developing a principled, secure-by-construction password manager.

Acknowledgements

We thank the anonymous reviews for their valuable feedback. We also thank Karthikeyan Bhargavan, David Wagner, Weichao Wang, Paul Youn, Chris Grier, Kurt Thomas, Matthew Finifter, Joel Weinberger, Chris Thompson, Suman Jana, and Nicholas Carlini for their valuable feedback and comments. This research was supported by Intel through the ISTC for Secure Computing; by the Air Force Office of Scientific Research (AFOSR) under MURI award FA9550-09-1-0539; by the Office of Naval Research (ONR) under MURI Grant N000140911081; and by the National Science Foundation (NSF) under grants 0831501CT-L and CCF-0424422. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, the AFOSR, the ONR, or Intel.

References

[1] B. Adida, A. Barth, and C. Jackson. Rootkits for javascript environments. In *Proc. of WOOT 2009*, 2009.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, 2010.

[3] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. In *Proc. the 21st USENIX Security symposium*, 2012.

[4] Ui attacks demos, 2013. `https://sites.google.com/site/webpwdmgr/`.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of ACM Conference on Computer and Communications Security*, 2008.

[6] A. Belenko and D. Sklyarov. "secure password managers" and "military-grade encryption" on smartphones: Oh, really?, 2012.

[7] K. Bhargavan and A. Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *Proc. of WOOT*, 2012.

[8] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security Symp.*, 2013.

[9] M. Blanchou and P. Youn. Password managers: Exposing passwords everywhere, Nov 2013. `https://www.isecpartners.com/media/106983/password_managers_nov13.pdf`.

[10] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. of IEEE Symp. on Security and Privacy*, 2012.

[11] CNET. Editor's rating of password managers. `http://download.cnet.com/windows/password-managers/?&sort=editorsRating+asc`.

[12] O. Connelly. *WordPress 3 Ultimate Security*. Packt Publishing Ltd, 2011.

[13] D. Crockford. Adsafe. *adsafe.org*, 2011.

[14] Content security policy: W3c editor's draft, 2013. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.

[15] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey, 2011.

[16] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[17] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, 2010.

[18] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2011.

[19] E. Grosse and M. Upadhyay. Authentication at scale. *Security Privacy, IEEE*, 11(1):15–22, Jan 2013.

[20] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proc. of NSPW*, 2009.

[21] A. Huth, M. Orlando, and L. Pesante. Password security, protection, and management. *United States Computer Emergency Readiness Team*, 2012.

[22] G. Inc. Google caja—google developers. `https://developers.google.com/caja/`.

[23] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational).

[24] Lastpass. `https://lastpass.com`.

[25] LastPass. Lastpass one million user giveaway. `http://blog.lastpass.com/2011/01/lastpass-one-million-user-giveaway.html`.

[26] X. Li and Y. Xue. A survey on server-side approaches to securing web applications. *ACM Computing Surveys*, 46(4), 2014.

[27] Z. Li, W. He, D. Akhawe, and D. Song. The emperor?s new password manager: Security analysis of web-based password managers. Technical Report UCB/EECS-2014-138, EECS Department, University of California, Berkeley, Jul 2014.

[28] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125–140, 2010.

[29] P. Magazine". Editor's rating of password managers. `http://www.pcmag.com/products/28042?sort=er+desc`.

[30] Needmypassword. `http://www.needmypassword.com`.

[31] Passwordbox. `https://www.passwordbox.com`.

[32] D. Pogue. Remember all those passwords? no need. `http://nyti.ms/10ZhXgq`, 2013.

[33] Roboform everywhere. `http://www.roboform.com/everywhere`.

[34] M. Rochkind. Security, forms, and error handling. In *Expert PHP and MySQL*, pages 191–247. Springer, 2013.

[35] D. Silver, S. Jana, E. Chen, C. Jackson, and D. Boneh. Password managers: Attacks and defenses. In *Proceedings of the 23rd Usenix Security Symposium*, 2014.

[36] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[37] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2011.

[38] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of ACM conference on Computer and communications security*, 2012.

[39] W3C. User interface safety directives for content security policy, 2012. `http://www.w3.org/TR/UISafety/`.

[40] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 365–379, 2012.

[41] C. Warren. Master passwords at risk in lastpass security breach. `http://mashable.com/2011/05/05/last-pass-breach/`.

[42] R. Woodbridge. "how passwordbox passed gmail as the #1 productivity app on their way to over 1m downloads". http://untether.tv/2013/episode-467, 2013.

[43] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.

SpanDex: Secure Password Tracking for Android

Landon P. Cox, *Duke University*
Peter Gilbert, *Duke University*
Geoffrey Lawler, *Duke University*
Valentin Pistol, *Duke University*
Ali Razeen, *Duke University*
Bi Wu, *Duke University*
Sai Cheemalapati, *Duke University*

Abstract

This paper presents SpanDex, a set of extensions to Android's Dalvik virtual machine that ensures apps do not leak users' passwords. The primary technical challenge addressed by SpanDex is precise, sound, and efficient handling of implicit information flows (e.g., information transferred by a program's control flow). SpanDex handles implicit flows by borrowing techniques from symbolic execution to precisely quantify the amount of information a process' control flow reveals about a secret. To apply these techniques at runtime without sacrificing performance, SpanDex runs untrusted code in a data-flow sensitive sandbox, which limits the mix of operations that an app can perform on sensitive data. Experiments with a SpanDex prototype using 50 popular Android apps and an analysis of a large list of leaked passwords predicts that for 90% of users, an attacker would need over 80 login attempts to guess their password. Today the same attacker would need only one attempt for all users.

1 Introduction

Today's consumer mobile platforms such as Android and iOS manage large ecosystems of untrusted third-party applications called "apps." Apps are often integrated with remote services such as Facebook and Twitter, and it is common for an app to request one or more passwords upon installation. Given the critical and ubiquitous role that passwords play in linking mobile apps to cloud-based platforms, it is paramount that mobile operating systems prevent apps from leaking users' passwords. Unfortunately, users have no insight into how their passwords are used, even as credential-stealing mobile apps grow in number and sophistication [12, 13, 24].

Taint tracking is an obvious starting point for securing passwords [11]. Under taint tracking, a monitor maintains a *label* for each *storage object*. As a process executes, the monitor dynamically updates objects' labels to indicate which parts of the system state hold secret information. Taint tracking has been extensively studied for many decades and has practical appeal because it can be transparently implemented below existing interfaces [11, 19, 5, 14].

Most taint-tracking monitors handle only *explicit flows*, which directly transfer secret information from an operation's source operands to its destination operands. However, programs also contain *implicit flows*, which transfer secret information to objects via a program's control flow. Implicit flows are a long-standing problem [8] that, if left untracked, can dangerously understate which objects contain secret information. On the other hand, existing techniques for securely tracking implicit flows are prone to significantly *overstating* which objects contain secret information.

Consider secret-holding integer variable s and pseudocode **if** s != 0 **then** $x := a$ **else** $y := b$ **done**. This code contains explicit flows from a to x and from b to y as well as implicit flows from s to x and s to y. A secure monitor must account for the information that flows from s to x and s to y, regardless of which branch the program takes: y's value will depend on s even when s is non-zero, and x's value will depend on s even when s is zero.

Existing approaches to tracking implicit flows apply static analysis to all untaken execution paths within the scope of a tainted conditional branch. The goal of this analysis is to identify all objects whose values are influenced by the condition. Strong security requires such analysis to be applied conservatively, which can lead to prohibitively high false-positive rates due to variable aliasing and context sensitivity [10, 14].

In this paper, we describe a set of extensions to Android's Dalvik virtual machine (VM) called SpanDex that provides strong security guarantees for third-party apps' handling of passwords. The key to our approach is focusing on the common access patterns and semantics of the data type we are trying to protect (i.e., passwords).

SpanDex handles implicit flows by borrowing tech-

niques from symbolic execution to precisely quantify the amount of information a process' control flow reveals about a secret. Underlying this approach is the observation that as long as implicit flows transfer a safe amount of information about a secret, the monitor need not worry about where this information is stored. For example, mobile apps commonly branch on a user's password to check that it contains a valid mix of characters. As long as the implicit flows caused by these operations reveal only that the password is well formatted, the monitor does not need to update any object labels to indicate which variables' values depend on this information.

To quantify implicit flows at runtime without sacrificing performance, SpanDex executes untrusted code in a data-flow defined sandbox. The key property of the sandbox is that it uses data-flow information to restrict how untrusted code operates on secret data. In particular, SpanDex is the first system to use constraint-satisfaction problems (CSPs) at runtime to naturally prevent programs from certain classes of behavior. For example, SpanDex does not allow untrusted code to encrypt secret data using its own cryptographic implementations. Instead, SpanDex's sandbox forces apps that require cryptography to call into a trusted library.

SpanDex does not "solve" the general problem of implicit flows. If the amount of secret information revealed through a process' control flow exceeds a safe threshold, then a monitor must either fall back on conservative static analysis for updating individual labels or simply assume that all subsequent process outputs reveal confidential information. However, we believe that the techniques underlying SpanDex may be applicable to important data types besides passwords, including credit card numbers and social security numbers. Experiments with a prototype implementation demonstrate that SpanDex is a practical approach to securing passwords. Our experiments show that SpanDex generates far fewer false alarms than the current state of the art, protects user passwords from a strong attacker, and is efficient.

This paper makes the following contributions:

- SpanDex is the first runtime to securely track password data on unmodified apps at runtime without overtainting or poor performance.
- SpanDex is the first runtime to use online CSP-solving to force untrusted code to invoke trusted libraries when performing certain classes of computation on secret data.
- Experiments with a SpanDex prototype show that it imposes negligible performance overhead, and a study of 50 popular, non-malicious unmodified Android apps found that all but eight executed normally.

The rest of this paper is organized as follows: Section 2 describes background information and our motivation, Section 3 provides an overview of SpanDex's design, Section 4 describes SpanDex's design in detail, Section 5 describes our SpanDex prototype, Section 6 describes our evaluation, and Section 7 provides our conclusions.

2 Background and motivation

Under dynamic information-flow tracking (i.e., taint tracking), a monitor maintains a *label* for each *storage object* capable of holding secret information. A label indicates what kind of secret information its associated object contains. Labels are typically represented as an array of one-bit *tags*. Each tag is associated with a different source of secret data. A tag is set if its object's value depends on data from the tag's associated source. *Operations* change objects' state by transferring information from one set of objects to another. Monitors *propagate tags* by interposing on operations that could transfer secret information, and then updating objects' labels to reflect any data dependencies caused by an operation. We say that information derived from a secret is *safe* if it reveals so little about the original secret that releasing the information poses no threat. However, if information is *unsafe*, then it should only be released to a trusted entity.

2.1 Related work: soundness, precision, and efficiency

The three most important considerations for taint tracking are soundness, precision, and efficiency. Tracking is *sound* if it can identify all process outputs that contain an unsafe amount of secret information. Soundness is necessary for security guarantees, such as preventing unauthorized accesses of secret information. Tracking is *precise* if it can identify how much secret information a process output contains. Precision can be tuned along two dimensions: better *storage precision* associates labels with finer-grained objects, and better *tag precision* associates finer-grained data sources with each tag.

Imprecise tracking leads to *overtainting*, in which safe outputs are treated as if they are unsafe. A common way to compensate for imprecise tracking is to require users or developers to *declassify* tainted outputs by explicitly clearing objects' tags.

Tracking is *efficient* if propagating tags slows operations by a reasonable amount. The relationship between efficiency and precision is straightforward: increasing storage precision causes a monitor to propagate tags more frequently because it must interpose on lower-level operations; increasing tag precision causes a monitor to do more work each time it propagates tags. Finding a suitable balance of soundness, precision, and efficiency

is challenging, and prior work has investigated a variety of points in the design space.

One approach to information-flow tracking is to use static analysis in combination with a secrecy-aware type system and programmer-defined declassifiers to prevent illegal flows [20]. This approach is sound, precise, and efficient but is not compatible with legacy apps. Integrating secrecy annotations and declassifiers into apps and platform libraries requires a non-trivial re-engineering effort by developers and platform maintainers.

An alternative way to ensure soundness is to propagate tags on high-level operations that generate only *explicit flows*. An explicit flow occurs when an operation directly transfers information from from a set of well-defined source objects to a set of well-defined destination objects [8]. For example, process-level monitors such as Asbestos [9], Flume [15], and HiStar [23] maintain labels for each address space and kernel-managed communication channel (e.g., file or socket), and propagate tags for each authorized invocation of the system API.

Such process-grained tracking is sound and efficient, but operations defined by a system API commonly manipulate fine-grained objects, such as byte ranges of memory. The mismatch between the granularity of labeled objects and operation arguments leads to imprecision. For example, once a process-grained monitor sets a tag for an address space's label, it conservatively assumes that any subsequent operation that copies data out of the address space is unsafe, even if the operation discloses no secret information.

As with language-based flow monitors, process-grained monitors must rely on trusted declassifiers to compensate for this imprecision. These declassifiers proxy all inter-object information transfers and are authorized to clear tags from labels under their control. However, because declassifiers make decisions with limited context, they can be difficult to write and require developers to modify existing apps.

Other monitoring schemes have improved precision by associating labels with finer-grained objects such as individual bytes of memory [5, 19]. While tracking at too fine a granularity leads to prohibitively poor performance [5, 19] (e.g., 10x to 30x slowdown), propagating tags for individual variables within a high-level language runtime is efficient [11]. The primary challenge for such fine-grained tracking is balancing soundness and precision in the presence of *implicit flows*.

As before, consider secret-holding variable s and pseudo-code **if** s != 0 **then** $x := a$ **else** $y := b$ **done**. Borrowing terminology from [18], we say that all operations between **then** and **done** represent the *enclosed region* of the conditional branch. Thus, the enclosed region contains explicit flows from a to x and from b to y. Operations like conditional branches induce implicit flows by transferring information from the objects used to evaluate a condition to any object whose value is influenced by an execution path through the enclosed region. We refer to the set of influenced objects as the *enclosed set*. The enclosed set includes all objects that are modified along the taken execution path as well as all objects that *might have been modified* along any untaken paths. To ensure soundness, a monitor must propagate s's tags to all objects in the enclosed set.

Propagating tags to members of the enclosed set can lead to overtainting in two ways. First, because a conditional branch does not specify its enclosed set, the membership must be computed through a combination of static and dynamic analysis [5, 18]. In our example, a simple static analysis of the program's control-flow graph could identify the complete enclosed set consisting of x and y. However, strong soundness guarantees require an overly conservative analysis of far more complex untaken paths containing context-sensitive operations and aliased variables. This can overstate which objects' values are actually influenced by a branch. Less conservative tag propagation creates opportunities for malicious code to leak secret information.

Second and more important, the amount of information transferred through a process' control flow is often very low. These information-poor flows expose the problem with tag imprecision. In particular, conventional monitors can only account for an implicit flow by propagating single-bit tags from the branch condition to members of the enclosed set. And yet members of the enclosed set can only reflect as much new information as the branch condition reveals. When the condition reveals very little information (e.g., s != 0), a single-bit tag cannot be used to differentiate between an object whose value is weakly dependent on secret information and one whose value encodes the entire secret. Thus, when an execution's control flow transfers very little information, propagating tags to members of the enclosed set significantly overstates how much secret information the branch transfers to the rest of the program state.

Prior work on DTA++[14] and Flowcheck [18] have articulated similar insights about the causes of overtainting. DTA++ propagates tags to an enclosed set only if an execution's control flow reveals the entire secret (i.e., the execution path is injective with respect to a secret input). However, DTA++ relies on offline symbolic execution of several representative inputs to select which branches should propagate tags to their enclosed sets. Offline symbolic execution provides limited code coverage for moderately complex programs and is unlikely to deter actively malicious programs.

Flowcheck focuses on the imprecision of single-bit taint tags and precisely quantifies the total amount of secret information an execution reveals (as measured in

bits). However, Flowcheck imposes significant performance penalties and must compute the enclosed set (often with assistance from the programmer) to quantify the channel capacity of enclosed regions.

To summarize, we are unaware of any prior work on information-flow tracking that provides a combination of soundness, precision, and efficiency that would be suitable for tracking passwords on today's mobile platforms.

2.2 Android-app study

To test our hypothesis that conventional handling of implicit flows leads to overtainting and false alarms, we created a modified version of TaintDroid [11] called TaintDroid++ that supports limited implicit-flow tracking. TaintDroid and TaintDroid++ track explicit flows the same way. Each variable in a Dalvik executable is assigned a label consisting of multiple tags, and tags are propagated according to a standard tag-propagation logic.

The primary difference between the two monitors is that TaintDroid ignores implicit flows and TaintDroid++ does not. First, for a Dalvik executable, TaintDroid++ constructs a control-flow graph and identifies the immediate post-dominator (ipd) for each control-flow operation. It then uses smali [1] to insert custom Dalvik instructions that annotate (1) each ipd with a unique identifier, and (2) each control-flow operation with the identifier of its ipd. Like Dytan [5], TaintDroid++ does not propagate tags to objects that might have been updated along untaken execution paths.

Using these two execution environments, we ran four popular Android apps that require a user to enter a password: the official apps for LinkedIn, Twitter, Tumblr, and Instagram. Both systems tagged password data as it was input but before it was returned to an app. We then manually exercised each app's functionality and monitored its network and file outputs for tainted data.

Figure 1 shows the number and type of tainted outputs we observed for apps running under TaintDroid and TaintDroid++. For each tainted output, we manually inspected the content to determine whether it contained password data or not. Each tainted output under TaintDroid appeared to be an authentication message that clearly contained a password. TaintDroid++ also tainted these outputs, but generated many more tainted network and file writes. We were unable to detect any password information in these extra tainted outputs, and regard them as evidence of overtainting.

Overtainting is only a problem if incorrectly tainted data is copied to an inappropriate sink. Thus, a false positive occurs when an app copies data that is safe but tainted to an inappropriate sink. Apps authenticate using the OAuth protocol and should not store a local copy

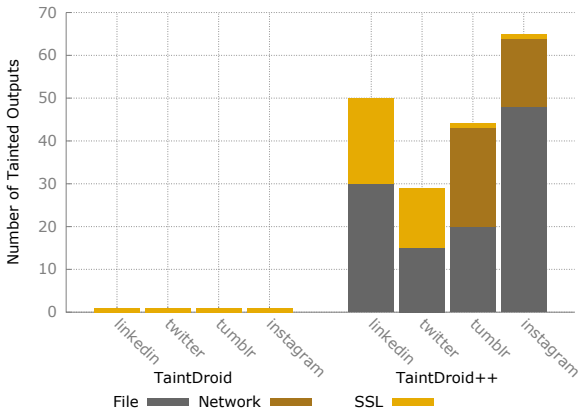


Figure 1: Tainted outputs for apps running under TaintDroid and TaintDroid++.

of a password once they receive an OAuth token from a server. Thus, each tainted file write generated under TaintDroid++ is a false positive.

For network writes, we also consider whether the password data was sent over an encrypted connection (i.e., over SSL) and the IP address of the remote server. Both Tumblr and Instagram under TaintDroid++ generated unencrypted tainted network writes. None of these writes were tainted under TaintDroid. Furthermore, TaintDroid only taints outputs to appropriate servers, but under TaintDroid++ several overtainted outputs were sent to third-parties such as the cloudfront.net CDN and flurry.com analytics servers. These results are consistent with previous work on overtainting [4, 22], and confirm that securing users' passwords requires a better balance of soundness and precision.

3 System Overview

This section provides an overview of SpanDex, including the principles and attacker model that inform its design.

3.1 Principles

SpanDex's primary goal is to soundly and precisely track how information about a password circulates through a mobile app. For example, if an app requests a Facebook password, then SpanDex should raise an alert only if the app tries to send an unsafe amount of information about the password to a non-Facebook server. Preventing leaks also requires a way for users to securely enter and categorize their passwords, and to address these issues we rely on secure password-entry systems such as ScreenPass [17]. SpanDex is focused on tracking information *after* a password has been securely input and handed over to an untrusted app. The following design principles guided our work.

Monitor explicit and implicit flows differently. In practice, explicit and implicit flows affect a program's state in very different ways. Operations on secret data that trigger explicit flows transfer a relatively large amount of secret information to a small number of objects. The inverse is true of control-flow operations that depend on secret data. These operations often transfer very little secret information to members of a large enclosed set. These observations led us to apply different mechanisms to tracking explicit and implicit flows.

First, SpanDex uses conventional taint tracking to monitor explicit flows. SpanDex is integrated with TaintDroid and Android's Dalvik VM, and maintains a label for each program variable. Each label logically consists of a single-bit tag indicating whether the variable contains an unsafe amount of information about a character within a user's password. Because explicit flows transfer a relatively large amount of information between objects, when an object's tag is set, SpanDex assumes that the variable contains an unsafe amount of secret information.

Second, when SpanDex encounters a branch with a tainted condition, it does not immediately propagate tags to objects in the enclosed set. Rather, SpanDex first updates an upper bound on the total amount of secret information the execution's control flow has revealed to that point. This upper bound precisely captures the maximum amount of secret information that an attacker could encode in untagged objects. As long as the total amount of secret information transferred through implicit flows is safe, SpanDex can ignore where that information is stored.

Like DTA++, SpanDex borrows techniques from symbolic execution to quantify the amount of information revealed through implicit flows. In particular, SpanDex integrates operation logging with tag propagation to record the chain of operations leading from a tainted variable's current state back to the original secret input. When SpanDex encounters a tainted conditional branch, it updates its information bounds by using these records to solve a constraint-satisfaction problem (CSP). The CSP solution identifies a set of secret inputs that could have led to the observed execution path. This set precisely captures the amount of information transferred through implicit flows.

The drawback of applying these techniques at runtime is the potential for poor performance. A monitor can efficiently record operations on tainted data at runtime, but solving a CSP when encountering a tainted branch could be disastrous. In the worst case, trying to solve a CSP could cause a non-malicious app to halt. For example, passwords must be encrypted before they are sent over the network, but it is infeasible to compute the set of all plaintext inputs that could have generated an encrypted output. Balancing the need to track implicit flows while preventing common primitives such as cryptography from slowing, or even halting, non-malicious apps led to our second design principle.

If commonly used functionality makes tracking difficult, force apps to use a trusted implementation. Mobile apps typically receive a password, perform sanity checks on the characters, encode the password as an http-request string, encrypt the http-request, and forward the encrypted string to a server. The code used to transform password data from one representation to another (e.g., encoding a character array as an http-request string and then encrypting the string) is problematic because it uses a number of operations that make quantifying implicit flows prohibitively slow or even impossible. This code includes a large number of bit-wise and array-indexing operations interleaved with tainted conditional branches. If SpanDex tracked implicit flows within this code as we have described thus far, non-malicious apps would become unusable.

Fortunately, it is exceedingly rare for apps to implement this functionality themselves. Instead, apps rely on platform libraries for common transformations, such as character encoding and cryptography. On Android this library code is small in size, easy to understand, and protected by the Java type system.

Tracking explicit flows remains the same for trusted libraries as for untrusted app code. However, within a trusted library, SpanDex does not solve CSPs when encountering a tainted branch and may directly update the information bound of a secret before exiting. This approach is sound for library code whose state is strongly encapsulated and whose semantics are well understood.

For example, encrypting a tainted string involves a sequence of calls into a crypto library for initializing the algorithm's state, updating that state, and retrieving the final encrypted result. Ignoring tainted conditional branches within this code is sound for two reasons. First, tracking explicit flows within the library ensures that any intermediate outputs as well as the final output are properly tagged. Second, external code can only access library state through the narrow interface defined by the library API; there is no way for untrusted code to infer properties about the plaintext except those that the library explicitly exposes through its interface or by branching on the plaintext data itself. SpanDex tracks both cases.

The protection boundary separating untrusted code from trusted library code has two novel properties. First, the boundary is defined by both data flow and control flow. An app is allowed to use a custom cryptographic implementation on untainted data, but must use the trusted crypto library to encrypt tainted data. Second, the boundary is enforced by the aggregate complexity of

the operations performed rather than by hardware or a conventional software guard. If an app attempts to encrypt password data using a custom implementation *or* branches on encrypted data returned by the trusted library, it will be forced to solve an intractable CSP and halt.

Thus, the key property of a SpanDex's sandbox is that it restricts the classes of computation that untrusted code may directly perform on secret data. Instead, an app must yield control to the trusted platform so that these computations can be performed on its behalf.

Given an execution environment that can efficiently quantify the amount of secret information transferred through implicit flows, SpanDex's final challenge is determining whether the quantified amount is safe to release. This challenge led to our final design principle.

Use properties of a secret's data type to set release policies. Like SpanDex, DTA++ requires a threshold on the amount of information revealed through implicit flows. DTA++ applies a strict policy to determine when to propagate tags by doing so only when the control flow is injective. That is, DTA++ propagates tags when a single secret value could have led to a particular execution path.

Though simple, this policy is inappropriate for SpanDex. Revealing an entire secret value via implicit flows is clearly unsafe, but revealing partial information about a password may be too. For example, using carefully crafted branches, malware could cause significant harm by narrowing every character of a password to two possible values. However, as we have seen, treating all implicit flows as unsafe leads to prohibitive overtainting. SpanDex's challenge is to support practical release policies that sit between these two extremes.

SpanDex benefits from its focus on passwords. Passwords have a well-defined representation and fairly well understood attacker model. For example, it is reasonable to assume that an attacker knows that a password consists of a sequence of human-readable characters (i.e., ASCII characters 32 through 126), many of which are likely to be alphanumeric. An attacker gains no new information from observing the control flow of a process if the flow reveals that each character is within the expected range of values. We investigate what apps' control flows reveal in Section 6.

3.2 Trust and attacker model

SpanDex is implemented below the Dalvik VM interface (i.e., the Dex bytecode ISA), and the protections provided by this VM provide the foundation for SpanDex's trust model. Most Android app logic is written in Java and compiled into Dex bytecodes, which run in an isolated Dalvik VM instance. SpanDex cannot protect passwords from an app that executes third-party native code *while there is password data in its address space*. Thus, objects tainted with password data must be cleared before an app is allowed to execute its own native code. In addition, once a process invokes third-party native code, it may not receive password data. SpanDex must rely on the kernel to maintain information about which processes have invoked third-party native code. Finally, apps may not write tainted data to persistent storage or send it to another app via IPC.

SpanDex is focused on securely tracking how password data flows within an app. Attacks on other aspects of password handling are outside the scope of our design. First, we assume that users can securely enter their password before it is given to an app, and that users will tag a password with its associated domain. A secure, unspoofable user interface, such as the one provided by ScreenPass [17], can provide such guarantees. Special purpose hardware, such as Apple's Touch ID fingerprint sensor and secure enclave [2], could also provide this guarantee.

Second, SpanDex can help ensure that password data is shared only with servers within the domain specified by the user, but provides no guarantees once it leaves a device. For example, SpanDex cannot prevent an attacker from sending a user's Facebook password as a message to a Facebook account controlled by the attacker. Preventing such cases requires cooperation between SpanDex and the remote server. SpanDex could notify the service when a message contains password data, and the service could determine whether such messages should contain password data.

We assume that an attacker completely controls one or more apps that a user has installed, and that the attacker is also in control of one or more remote servers. The attacker's servers can communicate with the attacker's apps, but the servers reside in a different domain than the one the user associates with her password. The attacker can make calls into the platform libraries and manipulate its apps' data and control flows to send information about passwords to its remote servers.

Based on the large-scale leakage of large password lists from major services, such as Gawker [21] and Sony Playstation [3], we assume that an attacker has access to a large list of unique passwords, and that the user's password is on the list. However, we assume that the attacker does not know which usernames are associated with each entry in its list (though it does know the user's username).

Thus, our attacker's goal is to de-anonymize the user within its password list using information gathered from its apps. The attacker can send its servers as much untainted data describing a user's password as SpanDex's release policies allow (i.e., the password length as well

as a range of possible values for each password character). In the worst case, the attacker will eliminate all but one of the passwords in its list. On the other hand, if the app provides no new information, then the user's password could be any on the list.

Once the attacker has computed the set of possible passwords for a username, it can only identify the correct username-password combination through online querying. For example, if an attacker infers that Bob's Facebook password is one of ten possibilities, then the attacker needs at most ten tries to login to Facebook as Bob.

The attacker may also have extra information about the usage distribution of passwords in its database. For example, the attacker may know that one password is used by twice as many users as another. While information from the app can help the attacker narrow a user's password to a smaller set of possibilities, the usage distribution allows the attacker to prioritize its login attempts to reduce the expected number of attempts before a successful login. We return to this issue in Section 6.

4 SpanDex

As with conventional taint tracking, SpanDex updates objects' labels on each operation that generates an explicit flow. If the monitor encounters a control-flow operation with a tainted condition, it does not update the labels of objects in the enclosed set. Instead, the monitor updates an upper bound on the amount of information the execution's control flow has revealed about the secret input.

SpanDex represents this bound as a *possibility set (p-set)*. SpanDex maintains a p-set for each password character an app receives. P-sets logically contain the possible values of a character revealed by a process' control flow. Each time the app's control flow changes as a result of tainted objects, SpanDex attempts to remove values from the secret's p-set.

4.1 Operation dag

In order to narrow a p-set, SpanDex must understand the data flow from the original secret values to a tainted condition. We capture these dependencies in an *operation dag (op-dag)*. This directed acyclic graph provides a record of all taint-propagating operations that influenced a tainted object's value as well as the order in which the operations occurred.

SpanDex reuses TaintDroid's label-storage strategy, and stores each 32-bit label adjacent to its object's value. However, whereas each bit in a TaintDroid label represents a different category of sensitive data (e.g., location or IMEI), SpanDex labels are pointers to nodes in the op-dag. If an object's label is null, then it is untainted. If an object's label is non-null, then its value depends on secret data.

Label storage in SpanDex most significantly differs from TaintDroid for arrays. In TaintDroid, each array is assigned a single label for all entries. If any array element becomes tainted, then the entire array is treated as tainted. This approach is inappropriate for SpanDex because we want to track individual password characters. Thus, SpanDex maintains per-entry labels. However, the reason that TaintDroid maintains a single label for each array is storage overhead. Byte and character arrays account for a large percentage of an app's memory usage, and assigning a 32-bit label for each byte-array entry could lead to a minimum fourfold increase in memory overhead for array labels.

To avoid this overhead, SpanDex allocates labels for arrays only after they contain tainted data. Each array is initially allocated a single label. If the array is untainted, then its label points to null. If the array contains tainted data, then its label points to a separate label array, with one label for each array entry. As with local-variable and object-field labels, array-element labels point to nodes in the op-dag. Since very few arrays contain password data, the overhead of maintaining per-entry labels is low overall.

The roots of the op-dag are special nodes that contain the original value of each secret (i.e., each password character), a pointer to the secret's p-set, and domain information. A p-set is represented as a doubly-linked list of value ranges. Each entry in the list contains a pointer to the previous and next entries, as well as a minimum and maximum value. Ranges are inclusive, and the union of the ranges specifies the set of possible secret values revealed by an app's control flow. SpanDex initializes p-sets to the range $[32, 126]$ to represent all printable ASCII characters. A secret's domain can be specified by the user through a special software keyboard [17].

Each tainted object version has an associated non-root node that records the operation that created the version, including its source operands. Source operands can be stored as concrete values (when operands are untainted) or as pointers to other nodes in the op-dag (when operands are tainted).

A node can point to more than one node, and there may be multiple paths from a node to one or more roots. The more complex the paths from a node to the op-dag roots are, the more complex updating p-sets becomes.

4.2 Example execution

If a tainted variable influences an app's control flow (e.g., via a conditional branch), then SpanDex traverses the op-dag from the node pointed to by the object's label toward

the roots. To demonstrate how SpanDex maintains and uses op-dags and p-sets, consider the simple snippet of pseduo-code below. Figure 2 shows the resulting op-dag and p-set.

```
0000: mov v1, v0          // v0, v1 label=ROOT
0002: add v2, v1, 3       // v2's label=N1
0004: add v2, v2, 2       // v2's label=N2
0006: sub v3, 6, v2       // v3's label=N3
0008: add v2, v2, 7       // v2's label=N4
000a: const/16 v4, 122    // v4's label=0
000c: if-le v3, v4, 0016
000e: ...
```

The first character of the password is 'p', or numeric value 112, and is stored in register v0. The password's domain is Facebook. v0's label points to the Root node for the secret character. v0 is then copied into v1, whose label must also point to Root. The sum of v1 and 3 is then stored in v2, whose label then points to new node, N1. N1 contains the addition operation, the 3 operand, and points to Root. The next line adds 2 to v2. This creates a new version of v2, which is recorded in N2. N2 contains the 2 operand and points to the node for the previous version of v2, node N1. The remaining arithmetic operations proceed similarly. Finally, the code loads the constant value of 122 into v4 for an upcoming conditional branch. v4's label is null, since it is not tainted.

When the code reaches the conditional branch, v3 is less than or equal to v4, since v3's value is 111, and v4's value is 122. Because v3's label is non-null, SpanDex uses the op-dag node in v3's label (N3) to update the p-set.

Updating the p-set is equivalent to solving a CSP to determine which secret values could have led to the control-flow change. In our example, updating the p-set is easy. SpanDex solves the inequality $v0+6-2-3 \leq 122$, leading to $v0 \leq 121$. Thus, the control flow reveals that the first character of the user's password is within the range of $[32, 121]$. SpanDex updates the p-set to reflect this before resuming execution. Figure 2 shows the state of the op-dag and p-set at this point.

This simple example demonstrates some of the challenges and nuances of SpanDex's approach. First, each node in the op-dag represents a version of a tainted variable. N3 points to the version of v2 used to update v3, so that when SpanDex reaches the conditional branch, it can retrieve the sequence of operations that led to v3's current value.

Second, reversible operations such as addition and subtraction make updating p-sets straightforward. Unfortunately, Dalvik supports a number of instructions that are much trickier to handle. For example, Dalvik supports instructions for operating on Java Object references and arrays that behave very differently than simple arithmetic operations. Even some classes of arithmetic operations, such as bit-wise operators and division, can make solving a CSP non-trivial.

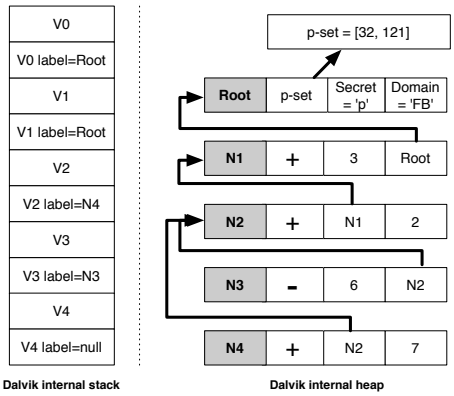


Figure 2: Simple op-dag and p-set example.

Third, there was a single path from N3 to Root in our example. If N3 had forked due to multiple tainted operands, or had led to multiple root nodes due to mixing secret characters, solving the CSP would have been far more complex. Compression and cryptography often mix information from multiple characters, which creates a complex nest of paths from nodes to the op-dag roots.

Fortunately, among the popular non-malicious apps that we have studied, difficult-to-handle operations occur only in platform code such as the Android cryptography library. Furthermore, it is rare to find apps that branch on the results of these operations outside of platform code. Thus, as long as SpanDex can ensure that all outputs from these libraries are explicit and tainted, then we can ignore implicit flows within them (and, thus, avoid CSP solving).

This approach is intuitive. First, outside of simple sanity checking on a password, there is little reason for an app to operate on password data itself. Second, libraries such as a crypto library are designed to suppress implicit flows. Observing an encrypted output or a cryptographic hash should not reveal anything about the plaintext input. Third, there is no obvious reason why app code should branch on either encrypted data or a cryptographic hash. Apps simply use the platform libraries to encode these outputs as strings and send them to a server.

There are many difficult operations that we have not observed in either app code or library code. Our general approach to these operations is to propagate taint to the results of these operations, but to fault if they cause the control flow to change. For example, an app may use bit-wise operations to encode a character, but branching on the encoded result is not allowed. This is secure and does not disrupt non-malicious apps. In the next section, we describe how SpanDex treats each class of Dex bytecodes in greater detail.

4.3 Dex bytecodes

In this section, we describe how SpanDex handles each of the following classes of bytecodes: type-conversion operations, object operations, control-flow operations, arithmetic operations, and array operations.

Type conversions. Dalvik supports the following data types: boolean, byte, char, short, int, long, float, double, and Object reference, as well as arrays of each of these types. P-set ranges are represented internally as pairs of floats. Solving CSPs involving conversions to alternate representations is supported as long as the type is a native and numeric.

Object operations. Dex provides a number of instructions for converting between data types, but conversions can also occur through Object-method invocations and arrays. For example, an app could index into an Object array with a tainted character, where the fields of each Object encodes its position in the array. The returned Object reference would be tainted, and would identify the character used to index into the array. The return value of any method used to access a field of the tainted Object would also be tainted. However, SpanDex would have to understand the internal semantics of the Object in order to solve a CSP involving the tainted returned value. Thus, branching on data derived from a tainted Object reference is not allowed.

Control-flow. A Dalvik program's control flow can change as a result of secret data in many ways. Conditional branch operations such as `if-eq` are the most straightforward, and SpanDex handles these as described in Section 4.2.

Dalvik also supports two case switching operations: `packed-switch` and `sparse-switch`. Both instructions take an index and a reference to a jump table as arguments. The difference between the instructions is the format of the jump table and how it is used. The table for a `packed-switch` is a list of key-target pairs, in which the keys are consecutive integers. Dalvik first checks to see if the index is within the table's range of consecutive keys. If it is not, then the code does not branch and execution resumes at the instruction following the switch instruction. If it is in the table, then the code computes the new PC by adding the matching target to the current PC.

The table for a `sparse-switch` is also a list of key-target pairs, but the keys do not have to be consecutive integers (though they have to be sorted from low-to-high). To handle this instruction, the VM checks whether the index is greater than zero and less than or equal to the table size. It then uses the index to perform a binary search on the keys to find a match. If it finds a match, then it jumps to the instruction at the sum of the matching target and current PC.

Although more complex than conditional branches, handling these switch instructions is straightforward. If the code falls through the switch instruction, then the resulting implicit flow reveals that the index is not equal to any of the table keys. SpanDex can solve a CSP for each of the keys and update its p-sets accordingly. If the control-flow is diverted by the switch instruction, then the resulting implicit flow reveals that the index is equal to the matching table key. SpanDex can solve a CSP for this condition as well. In practice, most switch instructions are packed and the corresponding jump tables are small, which makes solving CSPs for these operations fast.

Finally, a program's control flow can be influenced by tainted data if an operation on tainted data causes an exception to be thrown. For example, an app could divide a number by a tainted variable with a value of zero, or it could use a tainted variable to index beyond the length of an array. SpanDex could compute a CSP for the information revealed by each of these conditions, e.g., that a tainted variable is equal to zero or that a tainted variable is greater than the length of an array. However, we have not seen this behavior in any of the apps we have studied. As a result, our current implementation simply stops the program when an instruction with a tainted operand causes an exception to be thrown.

Arithmetic. As we saw in Section 4.2, reversible arithmetic operations are straightforward to handle. Other arithmetic operations are not impossible to handle, but require a complex solver. For example, reversing multiplication and division operations is tricky because of rounding. Bit-wise operations are even more difficult to reason about. Fortunately, it is exceedingly rare for app code to branch on the results of these operations. Instead, we have observed that trusted library code is far more likely to branch on the results of these operations. As long as we can ensure that all library outputs are explicit, then we do not need to solve CSPs involving difficult operations when in trusted code.

Arrays. Dex provides instructions for inserting (`iput`) and retrieving (`iget`) data from an array. Due to type-conversion problems, SpanDex does not allow tainted indexing of non-numeric arrays. In particular, an app may not use a tainted variable to index into an Object array.

Handling an `iget` operation requires keeping a checkpoint of the array in the op-dag node for the variable holding the result. For example, say that all of the entries in an int array are zero or one, and that an app indexes into the array with a tainted variable. The returned value would be stored in a tainted variable. If the app later branched on the tainted variable, then SpanDex must look at the array checkpoint to determine which indexes would have returned the same value as the exe-

cuted `iget`. In practice, tainted `iget` instructions are rare, and when they do occur the arrays are small.

Unlike a tainted `iget`, a tainted `iput` instruction is dangerous. Consider an attacker that initializes an array a with known size, such that all entries are equal to zero. It then stores the first password character in the variable s and inserts a one into $a[s]$. Because SpanDex maintains per-entry labels for arrays, $a[s]$ is tainted, but no other entries are. The attacker can then incrementally send each value in the array to its server: only $a[s]$ is tainted and will be stopped by SpanDex. Unfortunately, stopping the app at this point is too late, since the number of received zeros reveals the value of s. As a result of this attack, tainted `iput` instructions are illegal.

Finally, Dex also provides instructions such as `filled-new-array` for creating and populating arrays, and SpanDex disallows tainted operands on these instructions.

4.4 Trusted libraries

As described above, there are a number of operations on tainted data that would add significant complexity to SpanDex's CSP solver to support. Even worse, the complexity of the op-dags that combinations of these operations would create make it doubtful that even a sophisticated solver could handle them quickly, if at all. Ideally, these operations would never arise, and if they did, an app would never branch on their results. Sadly, this not the case. Many apps require cryptographic and string-encoding libraries to handle passwords, and these libraries are rife with difficult to handle operations as well as branching on the results of those operations.

Trying to solve such complex CSPs would make SpanDex unusable: non-malicious apps would halt just trying to encrypt a password. At the same time, ignoring flows generated by these operations is not secure. Luckily, we have observed that branching on the results of difficult operations consistently occurs within a handful of simple platform libraries.

Thus, SpanDex's approach to handling difficult implicit flows is to identify the functionality that creates them in advance and to isolate these flows inside trusted implementations. As long as the outgoing information flows from these libraries are always tainted and explicit, SpanDex does not need to worry about their internal control-flow leaking secret information. Furthermore, this code is open and well known, is protected by the Java type system, and can be modified to eliminate implicit flows through the library API.

The set of libraries that SpanDex trusts not to leak information implicitly is: java.lang.String (selected methods excluded), java.lang.Character, java.lang.Math, java.lang.IntegralToString, java.lang.RealToString, java.lang.AbstractStringBuilder, java.net.URLEncoder, java.util.HashMap, android.os.Bundle, android.os.Parcel, and org.bouncycastle.crypto. Nearly all of this code is either stateless string encoding and decoding or cryptography.

4.5 Various attacks and counter-measures

We described several attacks in Section 3.2 that are beyond the scope of SpanDex. In this section, we describe several other attacks and how SpanDex might handle them.

First, SpanDex does not allow tainted data to be written to the file system or copied to another process via IPC. This is reasonable because mobile apps should only require a user's password to retrieve an OAuth token from a remote server. After receiving the token, the app should discard the user's password. If an app tries to copy tainted data to an external server, then SpanDex must consult the domains in the set of reachable op-dag root nodes.

Second, an attacker could have multiple apps under its control generate multiple overlapping (but not identical) p-sets. Each individual p-set would appear safe, but when combined at the attacker's server, they could collectively reveal an unsafe amount of information. Relatedly, a malicious app could request a user's password multiple times and compute different ranges on each password copy.

One way to detect this class of attacks is by inspecting the membership of a secret's p-sets. For the apps that we have observed, p-sets usually correspond to natural character groupings, e.g., numbers, lower-case letters, upper-case letters, and related special characters. P-sets containing unusual character groupings could be a strong signal that an app is malicious.

The solution to this attack suggests a larger class of counter-measures that use information from the p-sets and op-dag to detect malicious behavior. For example, anomalous operation mixes or an unusually large op-dag could indicate an attack. One of the advantages of SpanDex is that it gives the monitor a great deal of insight into how an app operates on password data. We believe that this information could enable a rich universe of policies, though enumerating all of them is beyond the scope of this paper.

Finally, it is possible that SpanDex is vulnerable to certain classes of side-channel and timing attacks that we have not considered. However, any attack that relies on branching on tainted data would be detected. For example, consider the well-known attack on Tenex's password checker [16]. Even though the attack uses a page-fault side channel that is out of SpanDex's scope, SpanDex would have prevented it because each additional charac-

ter comparison would have narrowed its p-set to an unsafe level.

5 Implementation

Our SpanDex prototype is built on top of TaintDroid for Android 2.3.4. We modified TaintDroid to support p-sets and op-dags, and made several modification to the Android support libraries. Most of our changes to these libraries were made in java.lang.String.

First, public String methods whose return value could reveal something about a tainted string's value are not considered trusted to ensure that p-sets are updated properly (e.g., equals(Object), compareTo(String)).

Second, as a performance optimization, the Dalvik VM replaces calls to certain performance-critical Java methods with inlined "intrinsic" functions that are written in C and built in to the Dalvik VM (e.g., String.equals(Object), String.compareTo(String)). However, if an intrinsic inlined function operates on a tainted string and performs comparisons involving the string's characters, we are unable to update the p-sets accordingly. To avoid this, we modified Dalvik's intrinsic inlines that operate on strings to check if the string is tainted and, if so, invoke the Java version instead.

Third, Android's implementation of java.lang.String performs an optimization when converting an ASCII character to its String value: it uses the character's ASCII code to index into a constant char array containing all ASCII characters. If the character to be converted is tainted, we prevent this optimization from being used, as it would result in an array lookup with a tainted index.

Finally, we modified the android.widget.TextView and implemented a custom IME with a special tainted input mode that can be enabled to indicate to SpanDex when a sequence of characters is sensitive (i.e., a password).

6 Evaluation

In order to evaluate SpanDex, we sought answers to the following questions: How well does SpanDex protect users' passwords from an attacker? What is the performance overhead of SpanDex?

6.1 Password protection

As described in Section 3.2, we have designed SpanDex based on an attacker that has access to a large list of cleartext passwords. The attacker knows that a user's password is in the list, and uses untainted information from its malicious app to narrow a user's password to a smaller set of possibilities. To understand how well SpanDex can protect users from such an attack, we need to know the kind of p-sets that real apps induce, we need access to a large list of cleartext passwords, and we need a realistic distribution of how passwords are used. All of these pieces of information will allow us to calculate the number of expected logins an attacker would need to guess a user's password, given the amount of untainted password information that SpanDex allows apps to reveal.

First, we ran 50 popular apps from Google's Play Store. Each of these apps required a login, and we used the same 35-character password for each app. The password contained one lower-case letter ("a"), one upper-case letter ("A"), one number ("0"), and one of each of the 32 non-space special ASCII characters. 42 ran without modification[1]. The top row of Table 1 shows each character in the password.

Eight apps invoked native code before requesting a user's password[2]. While these apps would have to be modified to run under SpanDex, waiting to invoke native code before requesting a user's password is unlikely to require major changes. All other apps ran normally.

For the 42 apps that ran unmodified, after their password was sent, we inspected the p-set for each password character and counted its size. Table 1 shows the maximum, 75th percentile, median, 25th percentile, and minimum p-set size for each password character. The header of the table shows the password. The first thing to notice is that the p-sets for the letters in our password (i.e., "A" and "a") were never smaller than 26. This makes sense, since each app is branching to determine that the character is either a lower or upper case letter. The same is true for the number in our password, "0". No numeric p-set was smaller than 10.

The more difficult cases are the non-alphanumeric special characters. For these cases, the p-sets are fairly app specific. In some cases, the app's control flow depends on a specific character (e.g., Skout with several special characters), but most characters' p-sets remain large across most apps. With the exception of "*", "-", ".", and "_", all non-alphanumeric characters had large p-sets for 75% of apps or more.

Given this observed app behavior, we next obtained the uniqpass-v11 list of 131-million unique passwords [7]. The list contains passwords from a number of sources, including the Sony [3] and Gawker [21] leaks. To simulate an attack, we selected a password, p, from the list and computed the p-sets that a typical app would generate for p. In particular, we assume that the at-

[1] Audible, Amazon, Amazonmp3, Askfm, Atbat, Badoo, Chase, Crackle, Ebay, Etsy, Evernote, Facebook, Flipboard, Flixster, Foursquare, Heek, Howaboutwe, Iheartradio, imdb, LinkedIn, Myfitnesspal, Nflmobile, Pandora, Path, Pinger, Pinterest, Rhapsody, Skout, Snapchat, Soundcloud, Square, Tagged, Textplus, Tumblr, Tunein, Twitter, Walmart, Wordpress, Yelp, Zillow, Zite, and Zoosk

[2] Dropbox, Hulu+, Kindle, Mint, Skype, Spotify, Starbucks, and Voxer

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	:	;	<	=	>	?	@	A	[	\	]	^	_	`	a	{	|	}	~
Max	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95
75th	90	16	33	90	90	33	33	90	90	90	90	90	90	90	83	92	90	90	33	90	92	90	90	90	90	32	65	90	90	90	95	90	90	90	90
Med	16	12	12	16	16	12	12	16	16	13	16	16	13	13	14	10	7	7	7	7	7	7	7	26	6	6	6	6	6	6	90	4	4	4	4
25th	12	4	12	12	12	12	12	12	12	1	12	12	1	1	12	10	7	7	7	7	7	7	7	26	5	5	5	5	1	5	26	4	4	4	4
Min	1	1	3	1	1	1	1	1	1	1	3	4	1	1	1	10	1	1	1	1	1	1	3	26	1	1	2	4	1	4	26	1	1	3	4

Table 1: Password-character p-set sizes for 42 popular Android apps

tacker can infer p's length and whether each character is a lower-case letter (26 possibilities), an upper-case letter (26 possibilities), a number (10 possibilities), or a member of a block of special ASCII characters (i.e., the 16 characters below "0", the seven characters between "9" and "A", the six characters between "Z" and "a", and the four characters after "z").

This information gave us a kind of regular expression for p based on the type of each of its characters. We call the set of passwords matching this expression the *match set* and the size of the match set the *match count*. The larger a password's match count, the more uncertain an attacker is about what password the user entered. We computed the match count for all passwords in the uniqpass list in this way. Finally, we counted the number of passwords with a given match count to arrive at the inverse distribution function.

These calculations show that if SpanDex allows an attacker to learn the p-sets for a password from a typical app, the attacker will have trouble narrowing the set of possible passwords for the user. In particular, 92% of passwords have a match count greater than 10,000, 96% of passwords have a match count greater than 1,000, 98% of passwords have a match count greater than 100, and 99% of passwords have a match count greater than 10.

Unfortunately, recent work on a variety of password databases suggest that password usage follows a zipf distribution [6]. Thus, we also model the N passwords in a match set as a population of N elements that contains exactly one success (as a user would only have one correct password). Next, we let n be the random variable denoting the number of tries required to guess the correct password and find $E[n]$, the expected value of n. If the passwords are all equally probable, we try them in random order. Otherwise, we try them in the descending order of their probability. Note that each password try is done without replacement, i.e., after trying i passwords, we only consider the remaining $(N-i)$ passwords when picking the next most probable password.

A study of the distribution of passwords publicly leaked from Hotmail, flirtlife.de, computerbits.ie, and RockYou found that the passwords in each of these sets can be reasonably modelled by a zipf distribution with s parameter values of 0.246, 0.695, 0.23, and 0.7878 respectively [6]. Using these values of s, we modeled the passwords in each match set and computed the CDF of

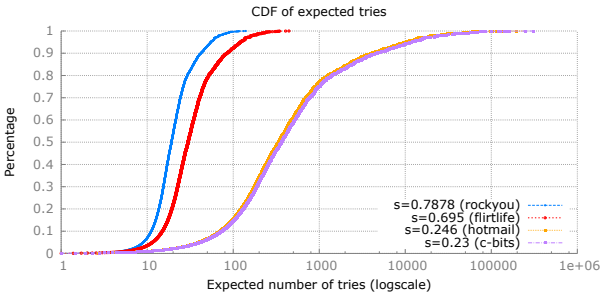


Figure 3: CDFs of expected login attempts using the uniqpass password list.

$E[n]$.

When $s = 0.7878$, 95% of the time, the attacker is likely to guess the correct password within 50 tries. When the s value for the zipf distribution is 0.246 or less, 99% of passwords are expected to require 10 or more login attempts, and 90% of passwords are expected to require 80 or more attempts. Figure 3 shows the CDFs for all four s values.

Unfortunately, we do not know the usage distribution for the uniqpass dataset since it contains only unique passwords.

6.2 Performance overhead

To measure the performance overhead of SpanDex we used the CaffeineMark benchmark and compared it to stock Android 2.3.4 and TaintDroid. Both TaintDroid and SpanDex ran without any tainted data. Since SpanDex only handles password data that is discarded after an initial login, this is SpanDex's common case. The benchmark was run on a Nexus S smartphone. The results are in Figure 4.

Overall, SpanDex performs only 16% worse than stock Android and 7% worse than TaintDroid. Stock Android performs significantly better than either TaintDroid or SpanDex in the string portion of the benchmark. This is because TaintDroid and SpanDex both disable some optimized string-processing code to store labels.

Finally, we would like to note that when testing apps in Section 6.1, we did not encounter any noticeable slow down under SpanDex. This was due to login being dominated by network latency and the simplicity of the CSPs these apps generated.

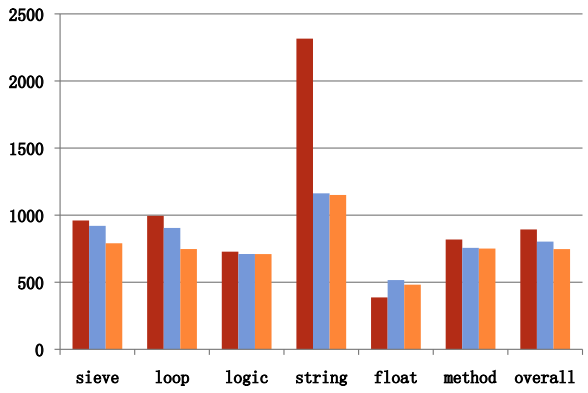


Figure 4: CaffeineMark results for Android (left bar), TaintDroid (middle bar), and SpanDex (right bar).

7 Conclusion

SpanDex tracks implicit flows by quantifying the amount of information transferred through implicit flows when an app executes a tainted control-flow operation. Using a strong attacker model in which a user's password is known to exist in a large password list, we found that for a realistic password-usage distribution, for 90% of users an attacker is expected to need 80 or more login attempts to guess their password.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. Our work was supported by Intel through the ISTC for Secure Computing at UC-Berkeley as well as the National Science Foundation under NSF awards CNS-0747283 and CNS-0916649.

References

[1] smali: An assembler/disassembler for android's dex format, 2013.

[2] Apple. iphone 5s, 2013.

[3] P. Bright. Sony hacked yet again, plaintext passwords, e-mails, dob posted, 2011.

[4] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA '08*, 2008.

[5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07*, 2007.

[6] David Malone and Kevin Maher. Investigating the Distribution of Password Choices. In *WWW '12*, April 2012.

[7] Dazzlepod. Uniqpass v11, 2013.

[8] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[9] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP '05*, 2005.

[10] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*, 2010.

[12] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys '12*, 2012.

[13] X. Jiang. Smishing vulnerability in multiple android platforms, 2012.

[14] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS '11*, 2011.

[15] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP '07*, 2007.

[16] B. W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17(5):33–48, Oct. 1983.

[17] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox. Screenpass: Secure password entry on touchscreen devices. In *MobiSys '13*, 2013.

[18] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. *SIGPLAN Not.*, 43(6):193–205, June 2008.

[19] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS '05*, 2005.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[21] L. Segall. Gawker data exposed in major hack attack, 2010.

[22] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *EuroSys '09*, 2009.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06*, 2006.

[24] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE SP '12*, 2012.

SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities

Yuchen Zhou David Evans
University of Virginia
[*yuchen*, *evans*]@*virginia.edu*
http://SSOScan.org

Abstract

Correctly integrating third-party services into web applications is challenging, and mistakes can have grave consequences when third-party services are used for security-critical tasks such as authentication and authorization. Developers often misunderstand integration requirements and make critical mistakes when integrating services such as single sign-on APIs. Since traditional programming techniques are hard to apply to programs running inside black-box web servers, we propose to detect vulnerabilities by probing behaviors of the system. This paper describes the design and implementation of SSOScan, an automatic vulnerability checker for applications using Facebook Single Sign-On (SSO) APIs. We used SSOScan to study the twenty thousand top-ranked websites for five SSO vulnerabilities. Of the 1660 sites in our study that employ Facebook SSO, over 20% were found to suffer from at least one serious vulnerability.

1 Introduction

Single Sign-On (SSO) services are increasingly used to implement authentication for modern applications. SSO-enabled applications allow users to log into an application using an established account (with a service such as Facebook or Twitter) and connect their account on the new site to an established Internet identity. Should the application need more information from the user, it may ask the user for extra permissions from the established service. Once granted, the requested information is returned to the application, which can then be used in the transparent account registration process.

Although these services provide SDKs intended to enable developers without security expertise to integrate their services, actually integrating security-critical third-party services correctly can be difficult. Wang et al. identified several ways applications integrating SSO SDKs can be vulnerable to serious attacks even when developers closely follow the documentation [27].

To better understand and mitigate these risks, we developed SSOScan, an automated vulnerability checker for applications using SSO. SSOScan takes a website URL as input, determines if that site uses Facebook SSO, and automatically signs into the site using Facebook test accounts and completes the registration process when necessary. Then, SSOScan simulates several attacks on the site while observing the responses and monitoring network traffic to automatically determine if the application is vulnerable to any of the tested vulnerabilities. We focus only on Facebook SSO in this work, but our approach could be used to check SSO integrations using other identity providers or other protocols. Many of our techniques could also be adapted to scan for vulnerabilities in integrating other security-critical services such as online payments and file sharing APIs.

1.1 Contributions

Our work makes two types of contributions: those related to the construction of our scanning tool which are largely independent of the particular vulnerabilities, and those resulting from our large-scale study of Facebook SSO implementations.

SSOScan. We explain the design and implementation of SSOScan (Section 3), as well as how to handle some of the challenges in the automation process. We describe techniques that automatically perform user interactions to walk through the SSO process (Section 3.1), including clicking the correct buttons and filling in registration forms. We collected information of almost 30,000 click attempts for sites that implement Facebook SSO which shows in detail how the individual heuristics are affecting SSOScan's behavior (Section 5.2). This provides experimental evidence to support our design choices and shed light on future research that shares a similar goal. SSOScan can detect whether a target application contains any of the five vulnerabilities listed in Section 2.2 with an

average testing time of 3.5 minutes, and is able to check 792 (81%) of the 973 websites that implement functional Facebook SSO from the top 10,000 with *no* human intervention at all.

Large-scale study. We ran SSOScan on the top 20,000 US websites (Section 4). Key results from the study include finding at least one vulnerability in 345 of the 1660 sites that use Facebook SSO (Section 4.1). We also learn how vulnerability rates vary due to different ways of integrating Facebook SSO (Section 4.1.1). We manually analyzed the 228 sites ranked in the top 10,000 that SSOScan cannot test automatically and report on the reasons for failures (Section 4.2). Our study reveals the complexity of automatically interacting with web sites that follow a myriad of designs, while suggesting techniques that could improve future automated testing tools. In Section 6, we discuss our experiences reporting the vulnerabilities to site owners and possible ways SSOScan could be deployed.

2 Background

This section provides a brief introduction to single sign-on systems, describes the vulnerabilities we checked, and summarizes relevant previous work.

2.1 Single Sign-On

A typical single sign-on process involves three parties. Alice first visits a web application and elects to use SSO to login. She is then redirected to the identity provider's SSO entry point (e.g., Facebook's server). After she logs into Facebook, her OAuth credentials are issued to the application server. The application server confirms the identity and authenticates the client.

OAuth uses three different types of (rather confusingly-named) credentials:

Access_token. An *access_token* represents permissions granted by the user. For example, the application may request that user grant permission to access the birthday and friend lists from her Facebook account. Upon the user's consent, a token will be issued and forwarded to the application which may then use it to obtain the granted information from Facebook. An *access_token* eventually expires, but may be valid for a long time.

Code. A *code* is used to exchange for an *access_token* through the identity provider. This exchange requires the application's unique *app_secret* to proceed. If the secret does not match, Facebook will not issue the token. This means a *code* is bound to a user as well as a target application. With Facebook SSO, the *code* expires after being used in the first exchange.

Signed_request. A *signed_request* is a base64 encoded string that contains a user identity, a *code*, and a signature that can be verified using an application's *app_secret* and some other metainformation. Once issued, it is not tied to Facebook (except for the enveloped *code*), and the signature can be verified locally.

2.2 Vulnerabilities

Our interest in building an automatic scanning tool was initially motivated by the *access_token* misuse vulnerability reported by Wang et al. [27]. We further identified four new vulnerabilities that are both serious and suitable for automatic testing. The first two vulnerabilities concern confusions about how authentication and authorization are done; the other three concern failures to protect important secrets.

Access_token misuse. This vulnerability stems from confusion about authentication and authorization. In OAuth 2.0, an *access_token* is intended for authorization purposes only because it is not tied to any specific application. When a service uses an *access_token* to authenticate users, it will also accept ones granted to any other application. Figure 1 illustrates an impersonation attack that exploits this vulnerability: Alice visits Mallory's website (step 1), logs in using Facebook SSO (2), and receives an *access_token* from Facebook (3). Then, Mallory's client-side code running in Alice's browser forwards the *access_token* to Mallory (4), which presents the token to a vulnerable application's server (5). After confirming the token represents Alice, Foo's application server authenticates Mallory as Alice (6).

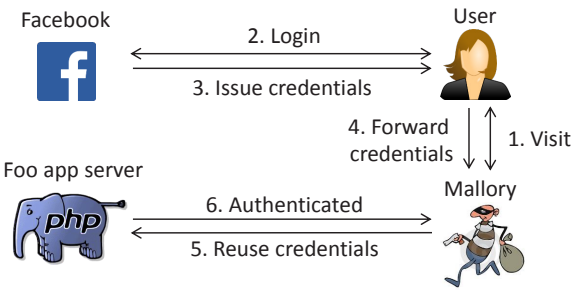


Figure 1: OAuth credential misuse

Signed_request misuse. Sometimes developers have chosen the correct OAuth credentials to use, but still end up with a vulnerable implementation. One way this happens is when information decoded from a *signed_request* is used but the signature is never checked using the *app_secret*. The attack to exploit this vulnerability is

similar to the previous one, except that Mallory needs to reuse the *signed_request* in addition to *access_token*.

App_secret leak. When a developer registers an application with Facebook, she receives an *app_secret*. It is essential for the application owner to keep it a secret because the *app_secret* is used as the key to create *signed_request*s and to access many other privileged functionalities. However, careless developers may reveal this secret to clients, especially when using *code* flow to authenticate users. By design, the *code* and *app_secret* must be sent from the application's back end server to Facebook in exchange for an *access_token*. When this exchange is carried out through the client instead of the server, *app_secret* is exposed to any malicious client.

User OAuth credentials leak. The last two vulnerabilities both leak a user's OAuth credentials. When the Facebook OAuth landing page contains third-party content, requests to retrieve those contents will automatically include OAuth credentials in the referer header, which leaks them to the third-party. To thwart this leakage, Facebook offers a layer of protection by only allowing *access_token* and *signed_request* to appear in the URL fragments, which are not visible in the referer header. Therefore only *code* can be leaked via referer unless the application intentionally pulls the credentials and puts it in the URL[1]. In addition, credentials can be exfiltrated by third-party scripts if they are present in the page content. If a malicious party is able to obtain these credentials, it could carry out impersonation attacks or perform malicious actions using permissions the user granted the original application, such as posting on the user's timeline or accessing sensitive information. Note the difference between embedding OAuth credentials in the URL and in the body content is that the former will directly leak them to third parties, while the latter only leaks the credential when the embedded third party code accesses it explicitly.

2.3 Related Work

Our work builds on extensive previous work on automatically testing applications for vulnerabilities. We briefly describe relevant approaches next, as well as previous works that analyze vulnerabilities in SSO services.

Program analysis. Program analysis techniques such as static analysis [3] and dynamic analysis including symbolic execution [7, 17] automatically identify vulnerabilities with fast testing speed and good code coverage. Runtime instrumentation techniques such as taint tracking [11] and inference [18] also help to safeguard sensitive source-sink pairs. However, these techniques require white-box access to the application (at least at the level of its binary), which is not available for remote web application testing. Automated web application testing tools that work on the server implementation [1, 8, 16] do not apply to large-scale vulnerability testing well. They either require access to application source code or other specific details such as UML or application states. For our purposes, the test target (application server implementation) is only available as a black box.

Automated security testing. Penetration testing is widely used to check applications for vulnerabilities [15, 28]. The tester analyzes the system and performs simulated attacks on it, often requiring substantial manual effort. More automated testing requires an oracle to determine whether or not a test failed. Sprenkle et al. developed a difference metric by comparing two webpages based on DOM structure and n-grams [21] and improved results using machine learning techniques [22]. SSOScan also requires an oracle (Section 3.2) to determine session identity. For our purposes, a targeted oracle works better than their generic approach.

Automated GUI testing. SSOScan is also closely related to automated GUI testing. The GUI element triggering approach we take shares some similarities with recent works to simulate random user interactions on GUI element to explore application execution space on Android system [14], native Windows applications [29], and web applications [5, 10]. Their common goal is to explore app execution space efficiently to discover buggy, abnormal or malicious behavior. By contrast, our goal is to drive the application through a particular SSO process rather than explore its execution space. Further, we need the tests to proceed fast enough for large-scale evaluation. Since each simulated user interaction with the web application involves round-trip traffic and a non-trivial delay to get the response, our primary focus is to develop useful heuristics to quickly prune search space before triggering any user interactions.

SmartDroid [32] and AppIntent [31] both aim to recover sequences of UI events required to reach a particular program state or follow an execution path obtained from static analysis. These approaches target Android applications and rely on client-side information that is not available for our web application scanning tool, where the necessary state only exists on the (inaccessible) server side.

Human cooperative testing. Off-the-shelf testing tools like Selenium [19] and TestingBot [24] can be used to discover bugs in web applications under developers' assistance. These tools replay user interactions based on testing scripts that are manually created by the applica-

[1] Surpsingly, we found several sites doing this (e.g., dealchicken.com and bloglovin.com).

tion developer. BugBuster [6] offers some automatic web application exploration capabilities, but still does not understand the application context enough to perform any non-trivial actions such as those involving authentication and business logic.

To reduce developer effort, Pirolli et al. [13], Elbaum et al. [9], and the Zaddach tool [12] show promising results by collecting interactions from normal users and replaying them to learn application states and invariants for vulnerability scanning. These works do not require extra manual effort from developers to write testing script or specify user interactions. However, one potential problem these works fail to address is user's privacy concerns when submitting interactions. This could be especially sensitive when the actions involve passwords or payments. SSOScan avoids this problem and is complementary to this line of work — SSOScan attempts to scan applications in a fully automatic fashion and does not require traces from any party.

Single sign-on security. Single sign-on has emerged as an important security service and has been well-studied in recent years. Previous works have discovered problems in protocols, bugs in SDK code and missed assumptions in developers' implementations [4, 20, 23, 25, 27]. Automated scanning is especially valuable for vulnerabilities that cannot be simply fixed by upgrading SDKs or improving the protocols, but stem from mistakes integrating the SSO service.

Integuard [30] and AuthScan [2] have similar goals with SSOScan. Integuard infers invariants across requests and responses and uses them to perform intrusion detection on future activities. AuthScan [2] is an automated tool to extract specifications from SSO implementations by using both static program analysis and dynamic behavior probing. Our goals differ in that we focus on detecting specific vulnerabilities rather than generic ones. This enables us to establish clear automation goals and build well-defined state machines for the scanner, and removes the uncertainties the previous works incur when inferring invariants or modeling unknown functions. The drawback is our approach relies on knowledge of particular vulnerabilities. For many integrated web services, including SSO, many vulnerabilities are known or can be obtained using systematic explication [27].

3 SSOScan

SSOScan consists of two main parts: the *Enroller* and the *Vulnerability Tester*. The Enroller automatically registers two test accounts at a web application using Facebook SSO. The Vulnerability Tester simulates attacks and monitors traffic to test for each vulnerability. In this section, we describe the general workflow of these modules necessary to understand the results in Section 4, but defer the details of our heuristics to Section 5.

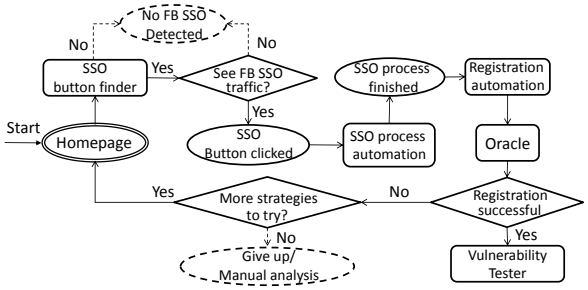


Figure 2: Enroller Overview.
Ovals represent testing states, curved rectangles represent different modules in our tool, and diamonds represent control flow decisions.

3.1 Enroller

Figure 2 shows the workflow of the Enroller. Given a target web application, our tool first removes all cookies from the browser and navigates to the target URL. A short delay after the page has fired its onload event, the SSO button finder (Section 3.1.1) analyzes the DOM and outputs the most likely candidate elements for SSO button. The Enroller then simulates clicks on those elements, monitoring traffic to listen for the Facebook SSO traffic pattern. Once a click or sequence of clicks is found that produces the recognizable SSO traffic, SSOScan automatically logs into Facebook and grants the requested permissions to the application.

About 44% of sites we tested still require a user to register when using SSO, so it is important to automate this process. SSOScan combines heuristics with random inputs to fill in and submit the forms (Section 3.1.2), and then uses an oracle (Section 3.2) to determine if the submission succeeds. If the oracle deems the registration to be a failure, the Enroller tries using different strategies (Section 5) until either the oracle passes or a threshold level of effort is exceeded. The entire process succeeds for 80% of the websites using Facebook SSO in the top 10,000 sites (Section 4 presents detailed results).

3.1.1 SSO Button Finder

A typical starting page, taken from huffingtonpost.com, is shown in Figure 3. SSOScan needs to first find and click the "Log in" button on the main page, and then the "Log in with Facebook" button on the overlay that pops up afterwards. As illustrated in Figure 4, SSOScan first extracts a list of qualifying elements from all nodes in an HTML page, and then extracts content strings from such elements. The Button Finder relies on the assumption that developers put one of a small pre-defined set

of expected words in the text content or attributes of the SSO button. It computes a score for each element by matching its content with regular expressions such as [Ll][Oo][Gg][liOo][Nn] which indicates its resemblance to "login". SSOScan forms a candidate pool consisting of the top-scoring elements and triggers clicks on them. (Section 5 describes the heuristic choices SSOScan uses to filter elements and compute scores.)

3.1.2 Completing Registration

The required interactions to complete the registration process after single sign-on vary significantly across web applications. They range from simply clicking a submit button (e.g., Figure 5, in which all input fields are pre-populated using information taken from the SSO process), to very complicated registration processes that involve interactively filling in multiple forms.

SSOScan attempts to complete all forms on the SSO landing page by leaving pre-populated fields untouched and processing the remaining inputs in the order of radios, selects, checkboxes and finally text inputs. We found this ordering to be very important to achieve higher automation success, as some forms may dynamically change what needs to be filled upon selecting different radio or select elements. Processing these elements first allows SSOScan to rescan for dynamically generated fields and process them accordingly.

For radio and select elements, SSOScan randomly chooses an option; for checkboxes, it simply checks all of them. For text inputs, SSOScan tries to infer their requirements using heuristics and provide satisfactory mock values. Once all the inputs have been filled, the next step is to reuse the SSO Button Finder (Section 3.1.1) with different settings designed to find submit buttons. After SSOScan attempts to click on a submit button candidate, it refers to the oracle to determine if the entire registration process is successful.

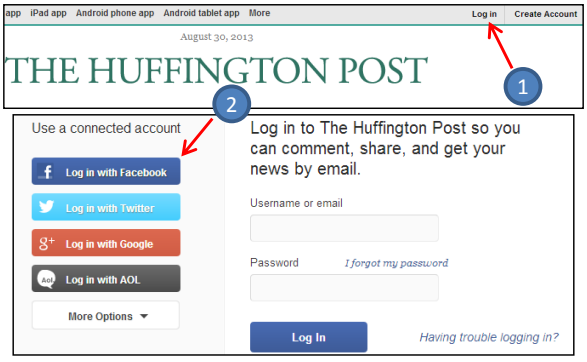


Figure 3: SSO Buttons (huffingtonpost.com)

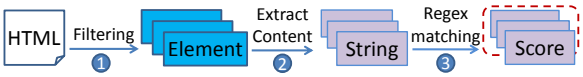


Figure 4: SSO button finder workflow

3.2 Oracle

The Oracle analyzes the application and determines whether it is in an authenticated state, and if so, further identifies the session identity. This module is necessary for SSOScan to decide if a registration attempt is successful. It is also used by the Vulnerability Tester to determine if a simulated impersonation attack succeeds.

The key observation behind the Oracle is that web applications normally remove the original login button and display some identifying information about the user in an authenticated session. For example, after a successful registration many websites display a welcome message that includes the user's first name.

After the page finishes loading, the Oracle searches the entire DOM and **document**.cookie for test account user information (e.g., names, email, or profile images). We evaluate the correctness of our assumptions and effectiveness of our Oracle in Section 4.2.

3.3 Vulnerability Tester

After the Enroller successfully registers two test accounts, control is passed to the Vulnerability Tester which checks the target application for the vulnerabilities described in Section 2.2. We use two different probing approaches to cover the five tested vulnerabilities: *simulated attacks* and *passive monitoring*.

Simulated Attacks. The two credential misuse vulnerabilities are tested using simulated impersonation attacks. We describe how this is done for *signed_request* misuses; the method for checking *access_token* misuses is similar.

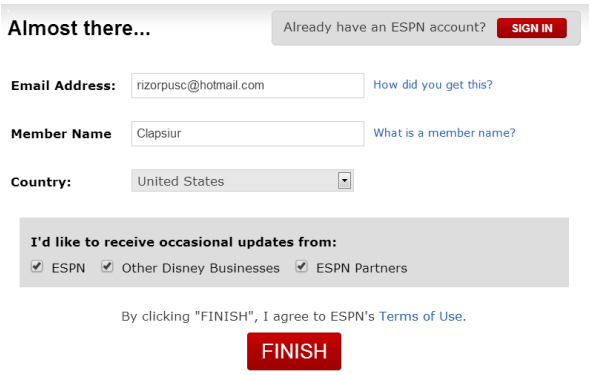


Figure 5: Registration Form (espn.go.com).

To set up the tests, we created a test application *Mal* which uses Facebook SSO, and obtained Alice's *signed_request* for *Mal*. This mimics the scenario where Alice is tricked into visiting and signing into an arbitrary malicious website using Facebook. After the account registration finishes, we use Bob's credentials to sign into Facebook for target application, but replace the *signed_request* in Facebook's response with the prior response received for Alice. For consistency, we also replace all *access_token*s found in the traffic.

The attack is successful if Bob is able to login as Alice using the replayed *signed_request*. The Vulnerability Tester deems the site vulnerable if the Oracle determines that Alice is logged in after the simulated attack.

Passive Monitoring. The three credential leakage vulnerabilities are detected using passive approaches. For brevity, we only explain how leaks through the referrer header are detected; the other leaks are detected similarly by observing network traffic and web page contents.

To check if an application leaks the user's OAuth credentials through the referrer header, SSOScan monitors all request data during the account registration process and compares each referrer header to OAuth credentials recorded in earlier stages. If a match is found, SSOScan then checks if the requesting page contains any third-party content such as scripts, images, or other elements that may generate an HTTP request. SSOScan reports a potential leakage when credentials are found in the referrer header for a page that contains third-party content.

4 Results

We evaluated SSOScan by running it on the list of the most popular 20,000 websites based on US traffic downloaded from quantcast.com as of 7 July 2013. Of those 20,000 sites, 715 of the sites are shown as *hidden profile* (that is, no URL is given, thus excluded from our study).

We ran SSOScan on the remaining 19,285 sites in September 2013, and found that homepages of 1372 sites failed to load during two initial attempts (most likely due to either expired or incorrect domain name, server error, or downtime). We excluded these sites from our data set, leaving a final test dataset containing 17,913 sites.

Completing the tests took about three days, running 15 concurrent sessions spread across three machines. The average time to test a site is 3.5 minutes. We limited the maximum stalling time for each site on any one module to four minutes, and the overall testing time to 25 minutes per site. If this timeout is reached, SSOScan restarts and retries a second time before skipping it. We ran extra rounds on tests that failed or stalled during initial round until either the test is completed or the four rounds maximum limit has been reached. The extra rounds involved fewer sites (<10%) and took a week to complete running on one machine with four concurrent sessions.

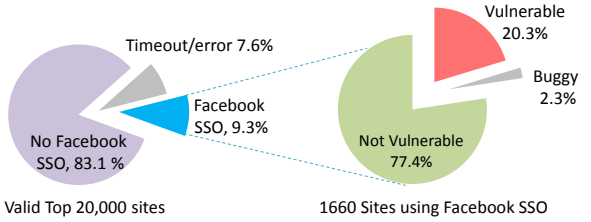


Figure 6: Results overview

In July 2014, we re-ran the tests on the vulnerable sites to see how many sites had corrected the vulnerabilities. The results from that scan are reported in Section 6.2.

4.1 Automated Test Results

Figure 6 presents results purely based on automatic tests run by SSOScan. SSOScan found a total of 1660 sites using Facebook SSO among the 17,913 sites (9.3% of the total). Figure 7 shows the number of Facebook SSO supported sites, sites that misuse credentials, and sites that leak credentials distributed by site ranking. The dotted lines on top of the bars show the average stats of all sites that are more popular than that rank. In Section 4.3, we report on our manual analysis on failed tests for sites ranked in the top 10,000.

Facebook SSO integration. Figure 7 (a) shows that more popular sites are more likely to integrate Facebook SSO. Of the top 1000 sites, 270 (27%) of them include Facebook SSO, compared to only 52 out of the 1000 lowest-ranked sites in our dataset. This supports our believe that covering the top-ranked 20,000 websites is sufficient to get a clear picture of prevailing Facebook SSO usage since less popular sites are both less visited and less likely to use Facebook SSO.

Faulty implementations. To implement Facebook SSO, an application must be configured correctly in the Facebook developer center. Using incorrect parameters to call the SSO entry point also result in errors that will prevent any user from authenticating to that application through SSO. Such cases, automatically identified by SSOScan, were more common than we expected. The most popular errors include setting the application to 'sandbox' mode (for development stage only) in the developer center, or providing a wrong application ID. SSOScan found 39 (2.3% out of 1660 sites that incorporate Facebook SSO buttons) sites that display visible Facebook SSO buttons but have implementations so buggy that no user could ever login using them. A possible explanation is that the buttons are there for SEO purposes and the developers

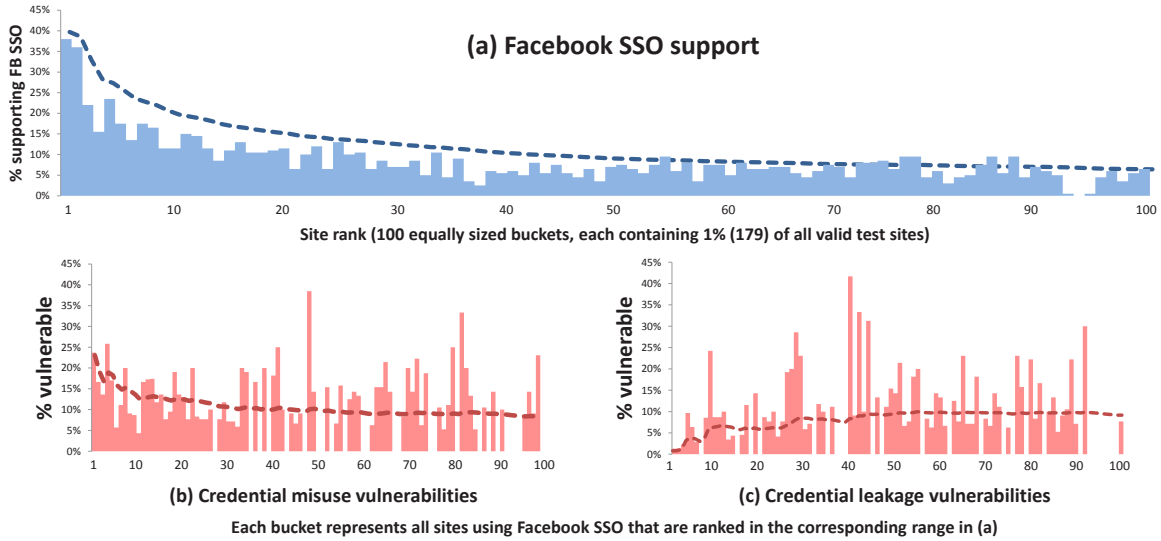


Figure 7: Facebook integration results by site rank

never actually bothered to implement it, or the developers simply copied and pasted an SSO snippet customized for another application without ever testing it.

Vulnerability trends. We found 202 sites (12.1%) that misuse credentials (126 of which are misusing both *access_token* and *signed_request*) and 146 sites (8.6%) that leak Facebook SSO credentials (of which 72 sites are leaking through both referrer headers and DOM). A total of 345 sites (20.3%) suffered from at least one of the five tested vulnerabilities, and 3 sites suffered from both credential misuse and leakage problems.

It is also worth noting that SSOScan did not find any sites leaking their *app_secret* to the public by calling the token exchange API on the client side. To verify that we implemented the check correctly, we have confirmed that SSOScan does correctly identify this vulnerability on our manually-crafted faulty application. This is an interesting result, especially compared to the high number of sites that have at least one of the other vulnerabilities. We suspect this is partly due to explicit warnings in the documentation and the increased effort required to actually implement the token exchange on the client side.

As shown in Figure 7 (b) and (c), more popular sites appear to be more likely to have credential misuse vulnerabilities, while less popular sites tend to have more credential leakage problems. This fact certainly raises concern — credential leakage could potentially do damage to users' Facebook accounts, and it would be hard to contact numerous low-profile problematic sites to have them all fixed. The victim's Facebook account is in jeopardy if any of the applications he or she uses have such problem. Even though credential misuse cannot harm Facebook accounts directly, the fact that such vulnerabilities exist in high-profile websites is worrisome, as impersonation attacks carried against sites with millions of users have more severe consequences thank similar attacks on lower-profile sites.

Of the top-1000-ranked sites, 60 of the 270 (22.2%) that support Facebook SSO are found to have at least one vulnerability. The vulnerability rate is 21.3% across all sites in the top 10,000 and 18.5% for sites ranked from 10,001 to 20,000. This overal vulnerability rate suggests that development practices at larger companies do not appear to be more stringent (at least with respect to SSO) than they are at less popular sites.

As we do not have access to server side source code, we cannot measure how reusing code may positively or negatively affect the vulnerability trend. However, we did notice that some sites use fourth party services (e.g. Janrain, Gigya) to implement the Facebook SSO. In such scenarios, the user effectively does two SSO processes during authentication — the user, Facebook (IdP) and Janrain (RP) initially; the user, Janrain (IdP) and the true relying party afterwards. As the Facebook SSO process is entirely handled by the fourth party and is hidden to the relying party, the RP's behavior is not relevant to this vulnerability. We have manually tested both Janrain and Gigya's Facebook SSO implementation for credential misuse vulnerabilities and confirmed that both of them correctly implement the process by only using *code* flow to authenticate users. As a result, sites using these services contribute to a lower vulnerability rate. Note that the RP would still need to implement the second SSO process correctly to avoid vulnerabilties, but SSOScan currently does not check IdPs other than Facebook.

4.1.1 Front-end Integration

There are three basic client-side methods to integrate Facebook SSO: a JavaScript SDK, a pre-configured widget, or a custom implementation. (We have no way to determine how the developers are integrating Facebook SSO at the back end.) We used SSOScan to aggregate front-end integration choices and compare them with vulnerability reports. Table 1 summarizes the results. Websites using client side SDKs and pre-configured widgets are more likely to misuse credentials (29.1% and 15.5% vs. 1.3% in non SDK/widget implementations). Our guess is that this is due to the way SDKs and widgets conveniently expose raw *access_token*, *signed_request*, or even user name Facebook ID values. This convenience may lead to the developers to neglect to check the signature and the intended audience of the credential. However, our results also show that websites using SDKs and widgets are better in hiding credentials (3.6% and 2.2% compared to 12.4% vulnerable rate in SDK/widget implementations). This is likely because such applications use the Facebook-provided landing page which has safe redirect URLs and no third-party content. Applications built this way are secure unless the developers explicitly add the credentials in the page content or URL.

4.1.2 Examples

We describe two examples of vulnerabilities found by SSOScan here to illustrate the potential risks. Section 6 discusses our experiences reporting vulnerabilities to site owners and Facebook.

Match.com. Ranked 118th on the list, Match.com is a popular online dating website. SSOScan revealed that match.com is also vulnerable to *signed_request* replacement attacks. To use match.com services, users need to provide sensitive information including their birthday, location, photos, personal interests, and sexual orientation. Impersonators will not only have access to this information, but also learn whom the victim is dating and possibly the time and location of the dates.

Fodors.com. Fodor's is a travel advice website that is the 217th-ranked US site. Its redirection landing page contains *access_token* information along with some other third-party scripts in its content. The scripts come from various sources including quantserve.com, fonts.com, yahooapis.com, and multiple domains owned by Google. The permission Fodor's requests includes user's basic information, email address, and more importantly, permission to post to user's wall on the his or her behalf. This means if the *access_token* is leaked to a malicious party, it can post to a user's Facebook wall without consent in addition to accessing the user's basic information.

Method	Number	Misuse	Leakage
SDK	578	29.1%	3.6%
Widget	132	15.5%	2.2%
Custom Code	950	1.3%	12.4%
All	1660	12.1%	8.6%

Table 1: Rate of credential misuse and credential leakage for different Facebook SSO front-end implementations

4.2 Detection Accuracy

To evaluate the detection accuracy of SSOScan, we sampled test cases from all results (including sites reported to have no Facebook SSO support, secure and vulnerable cases) and manually examined them. We consider two types of mistakes: misreporting whether the site integrates Facebook SSO, and incorrectly determining whether or not a Facebook SSO-enabled website exhibits a vulnerability.

Facebook Login Detection Correctness. SSOScan searches SSO button based on heuristics and cannot guarantee success for all websites. Indeed, it is not possible for anyone to determine with complete confidence if a website uses Facebook SSO by just browsing the site. To roughly measure how many Facebook SSO-enabled websites were missed by SSOScan, we randomly sampled the 100 sites that were reported by SSOScan to have no Facebook SSO support and manually examined them. To make the samples representative of the whole set, we picked one site out of every 200 sites ordered by their rank. From manually investigating these 100 sites, we could only find one site that included Facebook SSO but was missed by SSOScan. As we introduce later in Section 6, we also deployed SSOScan as a web service that is made available to use in our research group. The web service has received a total of 69 valid submissions so far and we have also manually examined the vulnerability reports.[2] We found four cases (5.8%) where a submitted site included Facebook SSO but SSOScan was not able to trigger it.

The sites that SSOScan fails to find Facebook login present unusual interfaces which our heuristics are not able to navigate to. Specifically, oovoo.com and bitdefender.com do not show any login button on its homepage, but instead the user needs to click a 'my account' button to initiate the login process. The sears.com site displays a login button on its homepage, but the SSO process is not initiated until the user interacts with the popup window three times, which exceeds the maximum

[2]These have mostly been sites suggested by people we have demoed SSOScan scan to, since the service has not yet been publicized. Hence, it is a small and non-representative sample, so not clear what we can conclude from this at this point.

click depths (two) in this evaluation. We have also seen one case (coursesmart.com) in which the login process is rather typical, but SSOScan still missed the correct login button (that button is scored the 4th highest while SSOScan only attempts to click the top 3 candidates.). Most of these issues may be addressed with more relaxed restrictions and more regular expression matching as described in Section 5.2. Finally, our prototype implementation is limited to English-language websites due to its string matching algorithm, but could be extended to include keywords in other languages.

SSOScan may also incorrectly conclude that a website supports Facebook SSO when it does not. We have seen sites (e.g., msn.com) that only use Facebook SSO to download user activities and display them on the page, but do not integrate their identity system with Facebook SSO. Although SSOScan is designed to skip searching on typical Facebook-provided social plugins and widgets, non-standard integration of such functionalities may rarely lead to false positives.

Vulnerability Status Correctness. Since SSOScan simulates potential attacks and verifies their success or failure, detection is likely to be highly accurate. Nevertheless, we consider several possible reasons that might cause false positives/negatives to be reported.

SSOScan should be able to capture all credential leakage vulnerabilities with no false positives. A false negative may occur since SSOScan only looks for exact matches to the original OAuth credential string, so it will not report a leakage if the credential is slightly transformed or encoded. Further, SSOScan only observes traffic involving the web client, so does not detect application that leak OAuth credentials outside the SSO process.

SSOScan only reports a credential misuse vulnerability when it can successfully execute an impersonation attack. So, the only risk for incorrect reports is if the Oracle incorrectly determines the session identity. We designed the Oracle to minimize this risk. For example, information for the test account is chosen carefully to be unlikely to appear otherwise but to be close enough to real names to pass sanity checks. For example, the randomly generated name "Syxvq Ldswpk" was rejected by a small number of websites, but "Jiskamesda Quanarista" always passed sanity checks and only appeared in an authenticated session in all of our tests. Barring an unlikely name collision, there does not appear to be any way SSOScan would produce a false positive credential misuse report.

The Oracle checks the whole response for identifying information instead of only the DOM content to handle sites which only embed such information in first-party cookies after logging in. In some rare cases, these cookies could be issued even before SSOScan finishes registration forms. This means that before the Enroller searches for registration forms to fill in, the Oracle deems registration as unnecessary because it concludes that the application is already in an authenticated state. Although SSOScan is able to proceed and determine vulnerability status, the application never enters an authenticated state and false negatives might occur.

Trusted Third-Party Domains. For credential leakage vulnerabilities, SSOScan reports an application as vulnerable if it identifies visible credentials co-existing with *any* content or script that comes from *any* origin other than the host or Facebook. This could overestimate the vulnerable sites because the host may own other domains and serve content over them, which should not be considered untrusted. For example, content delivery networks and sub-company scenarios (e.g., cnn.com embedding content from turner.com which owns CNN) are common among popular websites.

4.3 Automation Failures

For about 19% of the top 10,000 tested site that include functional Facebook SSO, SSOScan is not able to fully automate the checking process. Figure 8 shows the distribution of rank of failed test websites.

To better understand the reasons why SSOScan fails, we manually studied all 228 failed cases reported by SSOScan for sites ranked in the top 10,000. We found that although 47 out of these 228 cases set their Facebook application configurations and SSO entry points properly, they never respond to credentials returned by Facebook SSO, which means no users would be able to successfully log into these sites through Facebook SSO. Excluding these 47 left us a total of 181 failure cases.

Registration automation failure. By far the most common reason for SSOScan to fail is due to complicated or highly-customized registration process. We found 43.7%

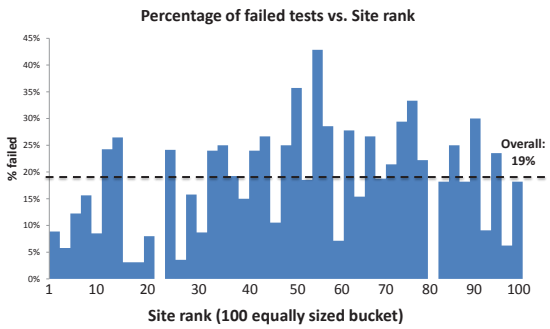


Figure 8: Failed tests rank distribution

Failure reason	Number	Percent
linking/subscription	51	28.1%
CAPTCHAs	34	18.8%
identity invisible to oracle	28	15.5%
atypical input elements	20	11.0%
atypical submit buttons	19	10.5%
email verification	10	5.5%
non-HTTPS submission forms	9	5.0%
other (e.g., timeouts)	10	5.5%
Total failures	181	100.0%

Table 2: Automation Failure Causes (top 10,000 sites)

of the sites that implement Facebook SSO still require users to perform additional actions to complete the registration (roughly evenly distributed by site popularity). SSOScan failed to complete registration on 143 (33.6%) of them. Table 2 shows the major reasons contributing to this failure ordered by their occurrences: 1) sites that require SSO users to link to an existing account or provide payment information to subscribe to the service; Currently SSOScan cannot handle the "linking" action: automatically registering a "traditional" account and perform the linking poses an out-of-scope challenge — doing so often requires solving CAPTCHAs[3]. 2) registration forms after the SSO process include CAPTCHAs; 3) special input elements (e.g. div, span or image as opposed to input) cannot be found automatically, or special requirements for the input that cannot be fulfilled; 4) sites where the registration submission button cannot be located; 5) sites that requires users to confirm email addresses before continuing (usually this involves clicking a link in an email sent by the server to the user's email address); and 6) sites that insecurely send registration data using a non-HTTPS form which causes the testing browser to pop up a warning and stall.

Oracle confusion. SSOScan may also fail because the oracle reports failure (15.5%), which occurs when it detects the login button no longer exists after Facebook SSO but cannot identify the session identity. We manually analyzed such cases and found the biggest obstacle is that the application homepage does not include any identifying information at all. For example, instead of showing 'Welcome, {username}', it shows 'Welcome, customer', or simply 'Welcome', and the user name is only displayed when accessing the account information page. In other cases, SSO authentication serves only a sub-service of the website such as its affiliated forum, but not the homepage which does not display any identifying information.

Others. During the testing, we have also seen a number of sites with extremely long loading time or inconsistent network latencies after Facebook SSO or upon navigating to certain pages. While the latency spikes can likely be resolved by re-running the tests, frequent long delays which accumulate to SSOScan's maximum timeout will always halt the automation process. For example, this happens when SSOScan accidentally triggers a browser confirmation dialog that requires user interaction, or asking users to stop a busy script execution.

[3]On the contrary, most tested applications (942 out of 973, see Section 4.3) do not ask users to solve CAPTCHAs when an account is created through SSO. This is a reasonable practice, since the user who is able to provide a valid Facebook account should have already passed Facebook's requirements, and adding additional CAPTCHAs would be unnecessarily annoying to the users.

5 Heuristics Evaluation

The ability of SSOScan to successfully complete the Facebook single sign-on and registration process depends on heuristics it uses to find buttons and fill in registration forms. Since each attempted button click involves a high-latency round-trip with the server, early pruning of search space and prioritization of elements is important for achieving successful completion within a reasonable amount of time. This section describes and analyzes the heuristics SSOScan uses. We analyze the click data collected from the top 10,000 sites that use Facebook SSO and show how tweaking the heuristics significantly improves performance.

5.1 Options

Each step in the automation process can be controlled by many options, including filters that can be enabled to eliminate candidate elements that are unlikely to be the correct target, weightings that adjust the contribution of different element properties to its score, and other behavior modifiers. The ones SSOScan used when running the Section 4 study are described below; additional options are described in our tech report [33].

Candidate rank. The button finder produces a candidate element list ranked by score. SSOScan will first attempt clicking on the highest-ranked element, but sometimes

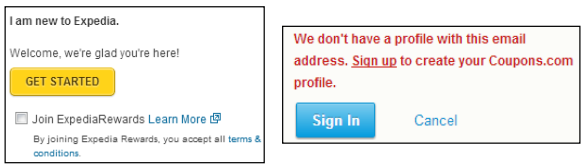


Figure 9: Example corner cases

the correct element is ranked lower. This option controls the maximum number of click attempts SSOScan makes before succeeding or giving up. For Section 4's experiment, the lowest ranked element SSOScan clicks is the third.

Visibility filter. Most websites only expect users to click on UI elements that are visible, so the button finder includes a filter that ignores all invisible elements (e.g., elements with zero height or width, shadowed in the background layer, or those which appear only when the user scrolls the initial screen position).

Position filter. We noticed that SSOScan sometimes gets distracted by a search box submit button when completing the registration form, even if it is able to correctly fill in the required information in all input elements. To eliminate these misclicks, the position filter eliminates the submit buttons which are displayed above any inputs based on our observation that submit buttons nearly always come last in a registration form.

Registration form filter. As mentioned earlier in Section 4.3, many websites provide two actions for the user after SSO is completed: 'create new account' or 'link an existing account'. The latter option requires the user to enter the user name and password of an existing account to finish the enrollment process. To avoid these, the registration form filter rejects a candidate submit button if its parent form contains only two visible text inputs, one has the meaning of 'name' or 'email' and the other is of type password, since such an element is most likely to be a submit button of a linking form.

Element content matching. SSOScan searches for elements whose labels are close to "login with Facebook" for SSO buttons by default. However, quite a few popular websites (e.g. coupons.com, right side of Figure 9) only allow users to "sign up with Facebook" first before logging in with Facebook. If the user has yet to do this, attempting to login with Facebook will produce an error. To handle this situation, SSOScan will search for elements with semantics similar to "sign up with Facebook" when it fails to register using the "login" buttons.

A filter may significantly reduce the number of misclicks. However, it may also occasionally exclude correct elements. For example, not every correct submit button is below all inputs (e.g., left of Figure 9, and expedia.com's submit button would have been missed with the element position filter enabled).

Hence, SSOScan is designed to explore target sites using different option settings if enrollment does not succeed with the initial settings. It will continue to attempt to complete the enrollment process using different settings until either all configurations have been exhausted or the timeout threshold is reached. SSOScan avoids doing duplicate work by detecting if a click attempt has resulted in a previously visited or completely explored state (see our tech report for details [33]).

5.2 Experiment Setup

In theory, SSOScan could exhaustively trigger clicks on every element on the page (and on all response pages up to some maximum depth), which would result in nearly 100% success rate. This would be prohibitively slow in practice, though, so the number of attempted clicks must be limited for any realistic test. Given the time needed for each click attempt, it is important to configure our scoring heuristics well to maximize the probability of a successful enrollment in the minimum amount of time.

To gather statistics about the candidate elements, we modified SSOScan to try all possible strategies even if it has already *found* the correct login button and to record information about all attempted clicks, including for example their size, position, visibility to the user, content string feature and whether it is *successful*. We define a click as *successful* if it is included in any sequence of clicks from the start page to triggering the SSO process, regardless of whether it appeared in an attempt that failed to trigger the process. Because SSOScan skips previously explored states to avoid redundant effort, it automatically rejects click sequences which involves cyclic state transitions such as clicking on an irrelevant link and then clicking on a logo which returns to the initial state.

We set up SSOScan to expand the candidate pool size for each configuration from 3 to 8, add more matching regular expressions (e.g., to match the string "forum" which occasionally leads to a login page on sites where no login is visible on the start page), and use equal weight for each of them. We also removed all candidate filters described in Section 5.1. Our goal is to capture as many ways to trigger the SSO process as possible by doing as close to an exhaustive search as is feasible. This increases the time required to scan a typical site to almost an hour (compared to a few minutes with the setup used in the full study).

We ran the test on the 973 sites from the top-10,000 ranked sites that were detected by SSOScan to support Facebook SSO in our main study (Section 4). This biases the study slightly, since it only includes sites where the configuration used in the initial study was able to find Facebook SSO. Ideally we would like to run all top-10,000 sites to avoid any bias introduced by the data set, but the significantly increased testing time prohibits us to do so, and the result of our subsequent study on a random sample of sites (Section 5.4) supports the claim that only few sites containing Facebook SSO were missed by the main study.

5.3 Results

The experiment recorded 29,539 unique[4] click attempts, of which 5086 (17.2%) are successful (that is, they either directly trigger SSO, or lead to subsequent clicks that trigger SSO). This amounts to approximately 30 unique clicks attempted per site, but the number varies significantly based on the site design, from a few up to 109.

Element type and content. Figure 10 shows how different button types and properties impact success rates. We report the success rate as the number of times that element appeared as a successful click divided by the total number of clicks attempted on elements of that type. The number beneath the element feature gives the total number of times that type of element occurred as a successful click target across all the test sites. For example, the *BUTTON* element type has an excellent success rate — 60% of all *BUTTON* candidates are true positives for the Facebook SSO button. But since it only appears as a successful click on 78 out of 973 sites in our sample, it is rarely useful. By contrast, clicking on *DIV* elements are much less likely to trigger the Facebook login, but such elements are more common. The right side of Figure 10 shows that elements that are directly visible to the user has a higher success rate than invisible ones, and elements residing in iframes are twice as likely to be the correct target as their counterparts in the main page. These results suggest ways of weighting element types to improve the scoring function and increase the likelihood of finding successful clicks early.

[4]If two clicks happens on pages with the same URL, same element XPath and same element outerHTML, we consider them the same click.

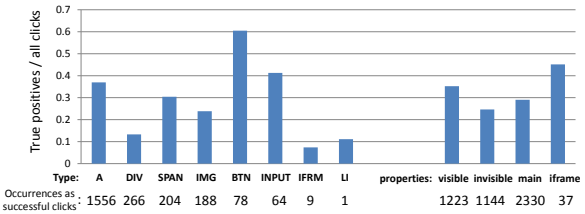


Figure 10: Login button type statistics

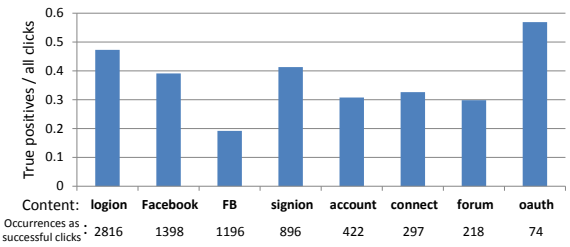


Figure 11: Login button content statistics

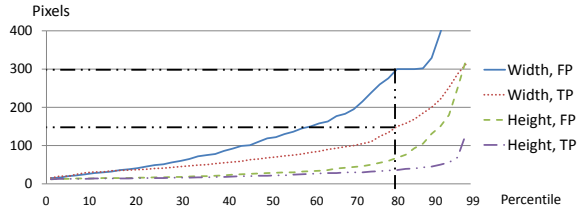


Figure 12: Impact of Login Button Size

Figure 11 shows how the success rate varies with attribute content (matched by the given regular expression). The keyword "oauth" rarely exists in any content, but when it appears it is very likely to identify the target element. The result also shows that "FB" is not a good indicator to predict the target, and we think this is probably because it is very short and may be used for similarly named JavaScript variables or random abbreviations.

Both figures include data for the *first* click only (but do measure first click success based on subsequent clicks). Data for the second clicks are noticeably different from the first, and overall success rates are lower on second clicks. The most interesting fact we found is that "connect" (39%) and "Facebook" (36%) become the most successful matches of all regular expressions, followed by "oauth" at (26%). No other regular expressions exceed 20% success for the second click.

Element size. Figure 12 gives the cumulative distribution function of the width and height of target elements. For example, the 80th percentile width of the true positive elements is approximately 150px, compared to 300px for false positive elements. We did not find any significant difference between first and second clicks, so the figure combines data from all clicks. The key result is that wide elements are less likely to be true positives, possibly due to SSOScan incorrectly including many large underlay elements as candidates. The result is similar for element height (the lower two lines in the figure). This suggests that it would be useful to add a filter function that excludes candidates whose width is greater than 300px. We would expect it to eliminate 20% of the false positives while hardly missing any of the true positives. Alternatively, SSOScan could adjust the final score of a node according to its size based on these results.

Element position. Figure 13 shows the heatmap of the login button's position in a page. The intensity at a location indicates the number of elements found there satisfying the property. Only visible elements are shown, and each successful click only attributes to the intensity once. All four figures are normalized with respect to their maximum intensity (i.e., element density).

The figures show an interesting distinction from first click to second click: successful first clicks almost ex-

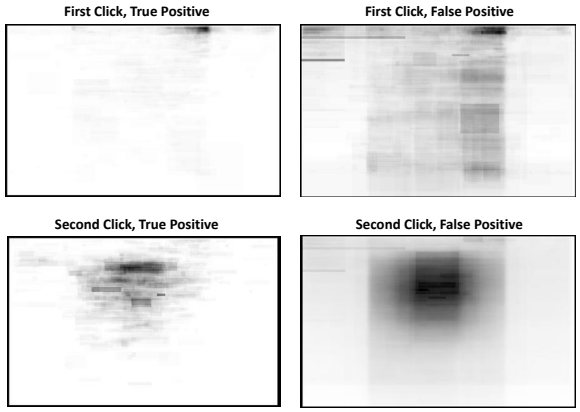


Figure 13: Login button location heatmap

clusively appear in the upper right corner of the page, while the second click appears generally in the upper-middle part of a page. The false positives are relatively more scattered everywhere on the page[5]. This result suggest we should assign a higher weight for elements for these locations, and focus on elements in the vicinity of the upper right corner for the first click. We could potentially even ignore the other criteria and only consider position to find login buttons on foreign-language sites.

5.4 Validation

After incorporating what we learned from these results (e.g., weight adjustment for different button sizes and types), we reran the SSOScan with the new heuristics on the sites ranked from 10,000 to 20,000 that SSOScan determined to support Facebook in the original study, which were not included in the heuristics evaluation. We compare the results with those obtained by using a "control" version of SSOScan, with equal weights on all features and no candidate filtering. All other settings such as candidate pool size are the same between two versions.

The results support the hypothesis that adjusting heuristics according to the results of the evaluation can improve the speed and robustness of detection of Facebook SSO integrations. The naïve control version missed 72 out of the 601 sites while the new heuristics missed only two. The average rank of correct candidate elements for the first and second click is 1.32 and 1.23 for the control experiment, which improves to 1.23 and 1.17 respectively with the new heuristics.

We also randomly picked 500 random sites from the sites that SSOScan have yet to find Facebook support in the experiment in Section 4. We tested the expanded heuristics on these sites, and further increased the maximum click depth to three to see if more SSO integrations could be found. Individual tests took an average of 31 minutes to finish, but varies significantly from a few minutes up to an hour (threshold) based on site content.

Four additional sites were found that support Facebook SSO from this sample in total. Two are found due to the added regular expression [Ff][Oo][Rr][Uu][Mm], one of which required three clicks to trigger the SSO process. Another site is found due to the improved candidate ranking algorithm, and the fourth was found using the new candidate selection method that includes all elements in the right corner of the page, even if they do not match any regular expressions. This provides a reasonable degree of confidence that our original study found a large enough fraction of all the popular sites using Facebook SSO to be representative, although likely missed around 1% of Facebook SSO sites. We did not try click depths greater than 3 because of the exponential time growth required to complete each test, but we feel confident that the number of Facebook SSO interfaces that can only be discovered by attempting more than 3 clicks is very low.

6 Discussion

This section concludes by discussing limitations of SSOScan, sharing our experiences reporting vulnerabilities, and suggesting ways SSOScan can be deployed to help secure applications integrating SSO services.

6.1 Limitations

While SSOScan is able to automatically synthesize basic user interactions and analyze traffic patterns, this approach is not suitable for detecting all types of vulnerabilities. It only works for vulnerabilities that can be checked by observing traffic or simulating predictable user events, and falls short if the vulnerability testing involves deep server-side application scanning or complicated interactions. For example, Wang et al. [27] point out that the application's *app_secret* might be leaked to arbitrary party if any page including Facebook's PHP SDK invokes two functions in a specific way. This type of vulnerability could be checked at the developer side using program analysis techniques, but cannot be checked by an external tool with no awareness of the sites' implementation details or internal state.

6.2 Communication and Responses

We started contacting the site owners shortly after obtaining our first list of vulnerable sites, manually sending out notifications to 20 vulnerable websites that we thought were interesting. We contacted them either by submitting

[5]The figures also show a clear width boundary. In the experiments the browser resolution is 1920x1200, and it seems that most developers' designs follow a standard width of approximately 960px, which is why the density appears to be cut off.

a form on their website or through email. The responses were very disappointing, especially compared with our previous experiences reporting SDK-level vulnerabilities to identity providers who tend to respond quickly and effectively to vulnerability reports [27]. The vulnerabilities found by SSOScan, on the other hand, are primarily in consumer-oriented sites without dedicated security teams or clear ways to effectively report security issues.

Of the 20 notifications, we only received eight responses, most of which appear to be automated. After the initial response, three websites sent us follow-up status updates. ESPN.com thanked us and told us the message has been passed onto appropriate personnel, but no follow up actions ensued. One of answers.com's engineers asked us for more details, but failed to respond again after we replied with proposed fix. As of July 2014, both sites are still vulnerable. Four months after getting the automated reply from ehow.com, we received a response stating that they have removed Facebook SSO from their website due to "content deemed inappropriate", and we have confirmed that the Facebook SSO button has indeed been removed. Sadly, we think their staff likely did not (bother to) understand our explanation for the fix and simply removed the feature.

The other instance where a reported vulnerability was fixed was for hipmunk.com. Hipmunk was found to be vulnerable to both the *access_token* and *signed_request* replacement attacks. We did not get any response from Hipmunk when the vulnerability was reported through the normal channels, but through a personal connection we were able to contact them directly. This led to a quick response and series of emails with one of Hipmunk's engineers. We explained how to check the signature of a *signed_request*, which should fix both vulnerabilities. However, when they got back to us believing that the fix was complete, we re-ran SSOScan and found that Hipmunk was still vulnerable to the *access_token* replacement attack. This meant Hipmunk checked the signature of *signed_request* after the fix, but never decoded the signed message body and compared its Facebook ID with the one returned by exchanging *access_token*. This surprised us, as we implicitly assumed the developers will consume the signed message body after verifying its signature, and thus only included 'verifying signature' in the proposed fix. After further explanation, the site was fixed and now passes all our tests.

Retesting vulnerable sites. We retested all 345 vulnerable sites in May 2014, nine months after our initial experiment, including the 20 websites we had notified directly. SSOScan found that 48 of the sites had eliminated the vulnerabilities (including one out of the 20 sites we contacted, mapquest.com). Of the 48 fixed sites, 22 had previously been diagnosed as credential leaking sites, and 27 were misusing credentials (one site, trove.com, fixed both problems). We further examined these sites manually to investigate the possible reasons and measures to fix the problems. As for sites that fixed credential misuses, we found that many had abandoned the *token* or *signed_request* flow in favor of the more secure *code* flow, which automatically protects them from credential reuse attacks. For credential leakages, we found that a number of sites redesigned their SSO process to feature a smoother user experience, e.g., replaced traditional redirection flows with AJAX operations, which naturally eliminated credential leakage via referer header.

Communication with Facebook. Due to the ineffectiveness of our direct communication with site owners, we contacted Facebook's platform integrity team in May 2014. Facebook's engineers indicated that they are particularly worried about *access_token* leakage through referer headers (because a malicious party in possess of the token may perform privileged Facebook actions on behalf of the user, which potentially directly harms Facebook), but are also concerned with the credential misuse scenario. Facebook asked for a list of the vulnerable applications and contacted all the sites with *access_token* leakage and credential misuse vulnerabilities (a total of 95 sites that we were able to re-confirm at the time of report), and informed us that they would "take enforcement action as necessary" upon the ten sites that are leaking *access_token*s in the referer headers. Facebook's engineers could not provide us with more information about what this entails or any direct responses they received, but an SSOScan re-run one month later (early July 2014) revealed that only four out of the 95 sites had fixed their problems (of the ten sites leaking *access_token*s, only two had been fixed). Even for Facebook, it appears to be difficult to convince consumer-focused websites to take security vulnerabilities seriously.

6.3 Deployment

Our experiences reporting vulnerabilities found by SSOScan suggest that notifying vendors individually will have little impact, which is consistent with experiences reported by Wang et al. with on-line stores [26]. Hence, we consider two alternate ways of deploying SSOScan to improve the security of integrated applications.

App center integration. We believe SSOScan would be most effective when used by an application distribution center (e.g. Apple store, Google Play) or identity provider (e.g., Facebook) as part of the application validation process. The identity provider has a strong motivation to protect users who use its service for SSO, and could use SSOScan to identify sites that can compromise those users. It could then deliver warning messages

to visitors of vulnerable applications during the log in through Facebook SSO process, or even go so far as to shut down SSO logins for that application. We also believe our results can provide guidance to vendors developing SSO services. The results in Section 4.1 indicate that sites are more likely to misuse credentials when using the Facebook JavaScript SDK. With Facebook's help, this problem could be mitigated by placing detailed instructions inside the SDK. The instructions could be presented as (non-executable) code in the SDK rather than as comments, so that the developers cannot get by without reading and removing them.

Checking-as-a-service. Without involving an centralized infrastructure, the best opportunity to deploy SSOScan is as a vulnerability scanning service that developers can use to check their implementations before their applications are launched (our prototype service at http://www.ssoscan.org/ can be used for this now). For a developer-directed test, it would be reasonable to ask the developer to either guide the tool through the registration process or provide a special test account that bypasses this step in cases where it cannot be fully automated. Even if we assume no aid from the developers, they should at least be able to tolerate a longer testing time than is feasible in doing a large-scale scan.

Availability

SSOScan is available at http://www.SSOScan.org/ as a public web service. The source code is available (linked from that site) under an open source license.

Acknowledgements

We thank Jonathan Burket, Longze Chen, Shuo Chen, Steve Huffman, Jaeyeon Jung, Haina Li, Chris Slowe, Ankur Taly, Rui Wang, Westley Weimer, Eugene Zarakhovsky and anonymous reviewers for their valuable inputs and constructive comments. This work has been supported by a Research Award from Google and research grants from the National Science Foundation and Air Force Office of Scientific Research.

References

[1] N. Alshahwan and M. Harman. Automated Web Application Testing Using Search Based Software Engineering. In *26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[2] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Suny, Y. Liuz, and J. S. Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *20th Network and Distributed System Security Symposium*, 2013.

[3] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *29th ACM Symposium on Principles of Programming Languages*, 2002.

[4] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *25th Computer Security Foundations Symposium*, 2012.

[5] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *11th International World Wide Web Conference*, 2002.

[6] BugBuster. BugBuster is a Software-as-a-Service to Test Web Applications. http://bugbuster.com/.

[7] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *12th International Conference on Model Checking Software*, 2005.

[8] G. Di Lucca, A. Fasolino, F. Faralli, and U. De Carlini. Testing Web applications. In *Journal of Software Maintenance*, 2002.

[9] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing with User Session Data. In *25th International Conference on Software Engineering*, 2003.

[10] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *12th International Conference on World Wide Web*, 2003.

[11] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Network and Distributed System Security Symposium*, 2007.

[12] G. Pellegrino and D. Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In *21st Network and Distributed System Security Symposium*, 2014.

[13] P. Pirolli, W.-T. Fu, R. Reeder, and S. K. Card. A User-tracing Architecture for Modeling Interaction with the World Wide Web. In *First Working Conference on Advanced Visual Interfaces*, 2002.

[14] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Third ACM Conference on Data and Application Security and Privacy*, 2013.

[15] Redspin Inc. Penetration Testing, Vulnerability Assessments and IT Security Audits. https://www.redspin.com/.

[16] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *23rd International Conference on Software Engineering*, 2001.

[17] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy*, 2010.

[18] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *16th Network and Distributed System Security Symposium*, 2009.

[19] Selenium development team. Selenium: Web application testing system. https://selenium.org/.

[20] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In *21st USENIX Security Symposium*, 2012.

[21] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated Replay and Failure Detection for Web Applications. In *20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[22] S. Sprenkle, E. Hill, and L. Pollock. Learning Effective Oracle Comparator Combinations for Web Applications. In *International Conference on Quality Software*, 2007.

[23] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *19th ACM Conference on Computer and Communications Security*, 2012.

[24] TestingBot. Selenium Testing in the Cloud - Run Your Cross Browser Tests in Our Online Selenium Grid. http://testingbot.com/.

[25] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *33rd IEEE Symposium on Security and Privacy*, 2012.

[26] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *32nd IEEE Symposium on Security and Privacy*, 2011.

[27] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *22nd USENIX Security Symposium*, 2013.

[28] Whitehat Security. Your Web Application Security Company. https://www.whitehatsec.com/.

[29] Q. Xie and A. M. Memon. Model-Based Testing of Community-Driven Open-Source GUI Applications. In *22nd IEEE International Conference on Software Maintenance*, 2006.

[30] L. Xing, Y. Chen, X. Wang, and S. Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *20th Network and Distributed System Security Symposium*, 2013.

[31] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *20th ACM Conference on Computer and Communications Security*, 2013.

[32] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[33] Y. Zhou and D. Evans. Technical Report: SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. https://www.ssoscan.org/SSOScan-TR.pdf.

When Governments Hack Opponents: A Look at Actors and Technology

William R. Marczak
UC Berkeley, Citizen Lab

John Scott-Railton
UCLA, Citizen Lab

Morgan Marquis-Boire
Citizen Lab

Vern Paxson
UC Berkeley, ICSI

Abstract

Repressive nation-states have long monitored telecommunications to keep tabs on political dissent. The Internet and online social networks, however, pose novel technical challenges to this practice, even as they open up new domains for surveillance. We analyze an extensive collection of suspicious files and links targeting activists, opposition members, and non-governmental organizations in the Middle East over the past several years. We find that these artifacts reflect efforts to attack targets' devices for the purposes of eavesdropping, stealing information, and/or unmasking anonymous users. We describe attack campaigns we have observed in Bahrain, Syria, and the United Arab Emirates, investigating attackers, tools, and techniques. In addition to off-the-shelf remote access trojans and the use of third-party IP-tracking services, we identify commercial spyware marketed exclusively to governments, including Gamma's FinSpy and Hacking Team's Remote Control System (RCS). We describe their use in Bahrain and the UAE, and map out the potential broader scope of this activity by conducting global scans of the corresponding command-and-control (C&C) servers. Finally, we frame the real-world consequences of these campaigns via strong circumstantial evidence linking hacking to arrests, interrogations, and imprisonment.

1 Introduction

Computer security research devotes extensive efforts to protecting individuals against indiscriminate, large-scale attacks such as those used by cybercriminals. Recently, the problem of protecting institutions against targeted attacks conducted by nation-states (so-called "Advanced Persistent Threats") has likewise elicited significant research interest. Where these two problem domains intersect, however—targeted cyber attacks by nation-states against *individuals*—has received virtually no significant, methodical research attention to date. This new problem space poses challenges that are both technically complex and of significant real-world importance.

In this work we undertake to characterize the emergent problem space of nation-state Internet attacks against individuals engaged in pro-democracy or opposition movements. While we lack the data to do so in a fully comprehensive fashion, we provide extensive detail from both technical and operational perspectives as seen in three countries. We view such characterizations as the fundamental first step necessary for the rigorous, scientific pursuit of a new problem space.

For our study we draw upon several years of research we have conducted into cases from Bahrain, Syria and the United Arab Emirates. We frame the nature of these attacks, and the technology and infrastructure used to conduct them, in the context of their impacts on real people. We hope in the process to inspire additional research efforts addressing the difficult problem of how to adequately protect individuals with very limited resources facing powerful adversaries.

As an illustration of this phenomenon, consider the following anecdote, pieced together from public reports and court documents.

At dawn on 3/12/13,[1] police raided the house of 17-year-old Ali Al-Shofa, confiscated his laptop and phone, and took him into custody. He was charged with referring to Bahrain's King as a "dictator" (الطاغية) and "fallen one" (الساقط) on a pseudonymous Twitter account, `@alkawarahnews`. According to court documents, Bahrain's Cyber Crime Unit had linked an IP address registered in his father's name to the account on 12/9/12. Operators of `@alkawarahnews` later forwarded a suspicious private message to one of the authors. The message was received on 12/8/12 on a Facebook account linked to the Twitter handle, and contained a link to a protest video, purportedly sent by an anti-government individual. The link redirected through `iplogger.org`, a service that records the IP address of anyone who clicks. Analytics for the link indicate that it had been clicked once from inside Bahrain. On 6/25/13, Ali was sentenced to one year in prison.

Ali's case is an example of the larger phenomenon we investigate: attacks against activists, dissidents, trade unionists, human rights campaigners, journalists, and members of NGOs (henceforth "targets") in the Middle East. The attacks we have documented usually involve the use of malicious links or e-mail attachments, designed to obtain information from a device. On the one hand, we have observed attacks using a wide range of off-the-shelf spyware, as well as publicly available third-party services, like `iplogger.org`. On the other hand, some attacks use so-called "lawful intercept" trojans and related equip-

[1] Dates in the paper are given MM/DD/YY.

ment, purportedly sold exclusively to governments by companies like Gamma International and Hacking Team. The latter advertises that governments need its technology to "look through their target's eyes" rather than rely solely on "passive monitoring" [1]. Overall, the attacks we document are rarely technically novel. In fact, we suspect that the majority of attacks could be substantially limited via well-known security practices, settings, and software updates. Yet, the attacks are noteworthy for their careful social engineering, their links to governments, and their real-world impact.

We obtained the majority of our artifacts by encouraging individuals who might be targeted by governments to provide us with suspicious files and unsolicited links, especially from unfamiliar senders. While this process has provided a rich set of artifacts to analyze, it does not permit us to claim our dataset is representative.

Our analysis links these attacks with a common class of actor: an attacker whose behavior, choice of target, or use of information obtained in the attack, aligns with the interests of a government. In some cases, such as Ali's, the attackers appear to be governments themselves; in other cases, they appear instead to be pro-government actors, ranging from patriotic, not necessarily skilled volunteers to cyber mercenaries. The phenomenon has been identified before, such as in Libya, when the fall of Gaddafi's regime revealed direct government ties to hacking during the 2011 Civil War [2].

We make the following contributions:

- We analyze the technology associated with targeted attacks (e.g., malicious links, spyware), and trace it back to its programmers and manufacturers. While the attacks are not novel—and indeed often involve technology used by the cybercrime underground—they are significant because they have a real-world impact and visibility, and are connected to governments. In addition, we often find amateurish mistakes in either the attacker's technology or operations, indicating that energy spent countering these threats can realize significant benefits. We do not, however, conclude that all nation-state attacks or attackers are incompetent, and we suspect that some attacks have evaded our detection.
- When possible, we empirically characterize the attacks and technology we have observed. We map out global use of two commercial hacking tools by governments by searching through Internet scan data using fingerprints for command-and-control (C&C) servers derived from our spyware analysis.
- We develop strong evidence tying attacks to government sponsors and corporate suppliers, countering denials, sometimes energetic and sometimes indirect, of such involvement [3, 4, 5, 6], in contrast to denials [7] or claims of a corporate "oversight" board [8]. Our scanning suggests use of "lawful intercept" trojans by 11 additional countries considered governed by "authoritarian regimes." We believe that activists and journalists in such countries may experience harassment or consequences to life or liberty from government surveillance.

Finally, we do not explore potential defenses appropriate for protecting the target population in this work. We believe that to do so in a sufficiently well-grounded, meaningful manner first requires developing an understanding of the targets' knowledge of security issues, their posture regarding how they currently protect themselves, and the resources (including potentially education) that they can draw upon. To this end, we are now conducting (with IRB approval) in-depth interviews with potential targets along with systematic examination of their Internet devices in order to develop such an understanding.

2 Related Work

In the past decades, a rich body of academic work has grown to document and understand government Internet censorship, including nationwide censorship campaigns like the Great Firewall of China [9, 10, 11]. Research on governmental Internet surveillance and activities like law-enforcement interception is a comparatively smaller area [12]. Some academic work looks at government use of devices to enable censorship, such as keyword blacklists for Chinese chat clients [13], or the Green Dam censorware that was to be deployed on all new computers sold in China [14]. We are aware of only limited previous work looking at advanced threat actors targeting activists with hacking, though this work has not always been able to establish evidence of government connections [15].

Platforms used by potential targets, such as GMail [16], Twitter [17], and Facebook [18] increasingly make transport-layer encryption the default, obscuring communications from most network surveillance. This use of encryption, along with the global nature of many social movements, and the role of diaspora groups, likely makes hacking increasingly attractive, especially to states who are unable to request or compel content from these platforms. Indeed, the increasing use of encryption and the global nature of targets have both been cited by purveyors of "lawful intercept" trojans in their marketing materials [1, 19]. In one notable case in 2009, UAE telecom firm Etisalat distributed a system update to its then 145,000 BlackBerry subscribers that contained spyware to read encrypted BlackBerry e-mail from the device. The spyware was discovered when the update drastically slowed users' phones [20]. In contrast to country-scale distribution, our work looks at this kind of pro-government and government-linked surveillance through highly *targeted* attacks.

The term APT (Advanced Persistent Threat) refers to a sophisticated cyber-attacker who persistently attempts to target an individual or group [21]. Work outside the academic community tracking government cyberattacks typically falls under this umbrella. There has been significant work on APT outside the academic community, especially among security professionals, threat intelligence companies, and human rights groups. Much of this work has focused on suspected government-on-government or government-on-corporation cyber attacks [22, 23]. Meanwhile, a small but growing body of this research deals with attacks carried out by governments against opposition and activist groups operating within, as well as outside their borders. One of the most notable cases is GhostNet, a large-scale cyber espionage campaign against the Tibetan independence movement [24, 25]. Other work avoids drawing conclusions about the attackers [26].

Country	Date Range	Range of Targets	Number and Type of Samples	Distinct Malware C&C's
Bahrain	4/9/12—7/31/13	$\geq$ 12 activists, dissidents, trade unionists, human rights campaigners, and journalists	8 FinSpy samples, 7 IP spy links received via private message, $>$ 200 IP spy links observed publicly	4 distinct IP addresses
Syria	2011 to present	10–20 individuals with technical backgrounds who receive suspect files from their contacts	40–50: predominantly BlackShades, DarkComet, Xtreme RAT, njRAT, ShadowTech RAT.	160 distinct IP addresses
UAE	7/23/12—7/31/13	7 activists, human rights campaigners, and journalists	31 distinct malware samples spanning 7 types; 5 distinct exploits	12 distinct IP addresses

Table 1: Range of data for the study.

Country	Possible Impacts	Probable Impacts
Bahrain	1. 3 individuals arrested, sentenced to 1–12 mo in prison 2. Union leader questioned by police; fired	1. Activist serving 1 yr in prison 2. Police raid on house
Syria	1. Sensitive opposition communications exposed to government 2. Exfiltrated material used to identify and detain activists	1. Opposition members discredited by publishing embarrassing materials 2. Exfiltrated materials used during interrogation by security services
UAE	Contacts targeted via malware	Password stolen, e-mail downloaded

Table 2: Negative outcomes plausibly or quite likely arising from attacks analyzed.

3 Data Overview and Implications

Our study is based on extensive analysis of malicious files and suspect communications relevant to the activities of targeted groups in Bahrain, Syria, and the UAE, as documented in Table 1. A number of the attacks had significant real-world implications, per Table 2. In many cases, we keep our descriptions somewhat imprecise to avoid potential leakage of target identities.

We began our work when contacted by individuals concerned that a government might have targeted them for cyber-attacks. As we became more acquainted with the targeted communities, in some cases we contacted targeted groups directly; in others, we reached out to individuals with connections to targeted groups, who allowed us to examine their communications with the groups. For Bahrain and Syria, the work encompassed 10,000s of e-mails and instant messages. For the UAE, the volume is several thousand communications.

4 Case Studies: Three Countries

This following sections outline recent targeted hacking campaigns in Bahrain, Syria and the UAE. These cases have a common theme: attacks against targets' computers and devices with malicious files and links. In some cases the attackers employed expensive and "government exclusive" malware, while in other cases, attackers used cheap and readily available RATs. Across these cases we find that clever social engineering often plays a central role, which is strong evidence of a well-informed adversary. We also, however, frequently find technical and operational errors by the attackers that enable us to link attacks to governments. In general, the attacks we find are not well-detected by anti-virus programs.

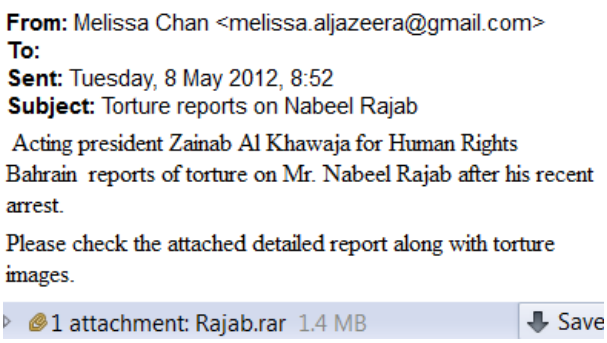


Figure 1: E-mail containing FinSpy.

4.1 Bahrain

We have analyzed two attack campaigns in the context of Bahrain, where the government has been pursuing a crackdown against an Arab-Spring inspired uprising since 2/14/2011.

The first involved malicious e-mails containing *FinSpy*, a "lawful intercept" trojan sold exclusively to governments. The second involved specially crafted *IP spy* links and e-mails designed to reveal the IP addresses of operators of pseudonymous accounts. Some individuals who apparently clicked on these links were later arrested, including Ali (cf. §1), whose click appears to have been used against him in court. While both campaigns point back to the government, we have not as yet identified overlap between the campaigns; targets of FinSpy appeared to reside mainly outside Bahrain, whereas the IP spy links targeted those mainly inside the country. We examine each campaign in turn.

FinSpy Campaign. Beginning in April 2012, the authors received 5 suspicious e-mails from US and UK-based activists and journalists working on Bahrain. We found that some of the attachments contained a PE (`.exe`) file designed to appear as an image. Their filenames contained a Unicode *right-to-left override* (RLO) character, causing Windows to render a filename such as `gpj.1bajaR.exe` instead as `exe.Rajab1.jpg`.

The other `.rar` files contained a Word document with an embedded ASCII-encoded PE file containing a custom macro set to automatically run upon document startup. Under default security settings, Office disables all unsigned macros, so that a user who opens the document will only see an informational message that the macro has been disabled. Thus, this attack was apparently designed with the belief or hope that targets would have reduced security settings.

Identification as FinSpy: By running the sample using Windows Virtual PC, we found the following string in memory: `y:\lsvn_branches\finspyv4.01\finspyv2\`. This string suggests FinSpy, a product of Gamma International [27]. The executables used virtualized obfuscation [28], which appeared to be custom-designed. We devised a fingerprint for the obfuscater and located a structurally similar executable by searching a large malware database. This executable contained a similar string, except it identified itself as `FinSpy v3.00`, and attempted to connect to `tiger.gamma-international.de`, a domain registered to Gamma International GmbH.

Analysis of capabilities: We found that the spyware has a modular design, and can download additional modules from a command & control (C&C) server, including password capture (from over 20 applications) and recording of screenshots, Skype chat, file transfers, and input from the computer's microphone and webcam.

To exfiltrate data back to the C&C server, a module encrypts and writes it to disk in a special folder. The spyware periodically probes this folder for files that match a certain naming convention, then sends them to the C&C server. It then overwrites the files, renames them several times, and deletes them, in an apparent effort to frustrate forensic analysis.

Analysis of encryption: Because the malware employed myriad known anti-debugging and anti-analysis techniques, it thwarted our attempts to attach debuggers. Since it did not include anti-VM code, we ran it in TEMU, an x86 emulator designed for malware analysis [29]. TEMU captures instruction-level execution traces and provides support for taint-tracking.

We found that FinSpy encrypts data using a custom implementation of AES-256-CBC. The 32 byte AES key and 16 byte IV are generated by repeatedly reading the low-order-4-bytes of the Windows clock. The key and IV are encrypted using an embedded RSA-2048 public key, and stored in the same file as the data. The private key presumably resides on the C&C server. The weak AES keys make decryption of the data straightforward. We wrote a program that generally can find these keys in under an hour, exploiting the fact that many of the system clock readings occur within the same clock-update quantum.

In addition, FinSpy's AES code fails to encrypt the last block of data if less than the AES block size of 128 bits, leaving trailing plaintext. Finally, FinSpy's wire protocol for C&C communication uses the same type of encryption, and thus is subject to the same brute force attack on AES keys. While we suspect FinSpy's cryptographic deficiencies reflect bugs, it is also conceivable that the cryptography was deliberately weakened to facilitate one government monitoring the surveillance of others.

C&C server: The samples communicated with `77.69.140.194`, which belongs to a subscriber of Batelco, Bahrain's main ISP. Analyzing network traffic between our infected VM and the C&C server revealed that the server used a global IPID, which allowed us to infer server activity by its progression.

In response to our preliminary work an executive at Gamma told the press that Bahrain's FinSpy server was merely a proxy and the real server could have been anywhere, as part of a claim that the Bahrain FinSpy deployment could have been associated with another government [4]. However, a proxy would show gaps in a global IPID as it forwarded traffic; our frequent observation of strictly consecutive IPIDs thus contradicts this statement.

Exploitation of captured data: Since we suspected the spyware operator would likely seek to exploit captured credentials, particularly those associated with Bahraini activist organizations, we worked with *Bahrain Watch*, an activist organization inside Bahrain. Bahrain Watch established a fake login page on their website and provided us with a username and password. From a clean VM, we logged in using these credentials, saving the password in Mozilla Firefox. We then infected the VM with FinSpy and allowed it to connect to the Bahrain C&C server. Bahrain Watch's website logs revealed a subsequent hit from `89.148.0.41`—made however to the site's homepage, rather than its login page—coming shortly after we had infected the VM. Decrypting packet captures of the spyware's activity, we found that our VM sent the password to the server exactly one minute earlier:

```
INDEX,URL,USERNAME,PASSWORD,USERNAME FIELD,
PASSWORD FIELD,FILE,HTTP 1,
http://bahrainwatch.org,bhwatch1,watchba7rain,
username,password,signons.sqlite,,
Very Strong,3.5/4.x
```

The URL provided to the server did not include the path to the login page, which was inaccessible from the homepage. This omission reflects the fact that the Firefox password database stores only domain names, not full login page URLs, for each password. Repeating the experiment again yielded a hit from the same IP address within a minute. We inspected Bahrain Watch's logs, which showed no subsequent (or previous) activity from that address, nor any instances of the same User Agent string.

IP spy Campaign. In an *IP spy* attack, the attacker aims to discover the IP address of a victim who is typically the operator of a pseudonymous social media or e-mail account. The attacker sends the pseudonymous account a link to a webpage or an e-mail containing an embedded remote image, using one of many freely-available services.[2] When the victim clicks on the link or opens the e-mail, their IP address is revealed to the attacker.[3] The attacker then discovers the victim's identity from their ISP. In one case we identified legal documents that provided a circumstantial link between such a spy link and a subsequent arrest.

Figure 2 illustrates the larger ecosystem of these attacks. The attackers appear to represent a single entity, as the activity all connects back to accounts that sent links shortened using a particular user account *al9mood*[4] on the `bit.ly` URL shortening service.

Recall Ali Faisal Al-Shufa (discussed in Section 1), who was accused of sending insulting tweets from an account

[2] e.g., `iplogger.org`, `ip-spy.com`, `ReadNotify.com`.

[3] Several webmail providers and e-mail clients take limited steps to automatically block loading this content, but e-mails spoofed to come from a trusted sender sometimes bypass these defenses.

[4] A Romanization of the Arabic word for "steadfastness."

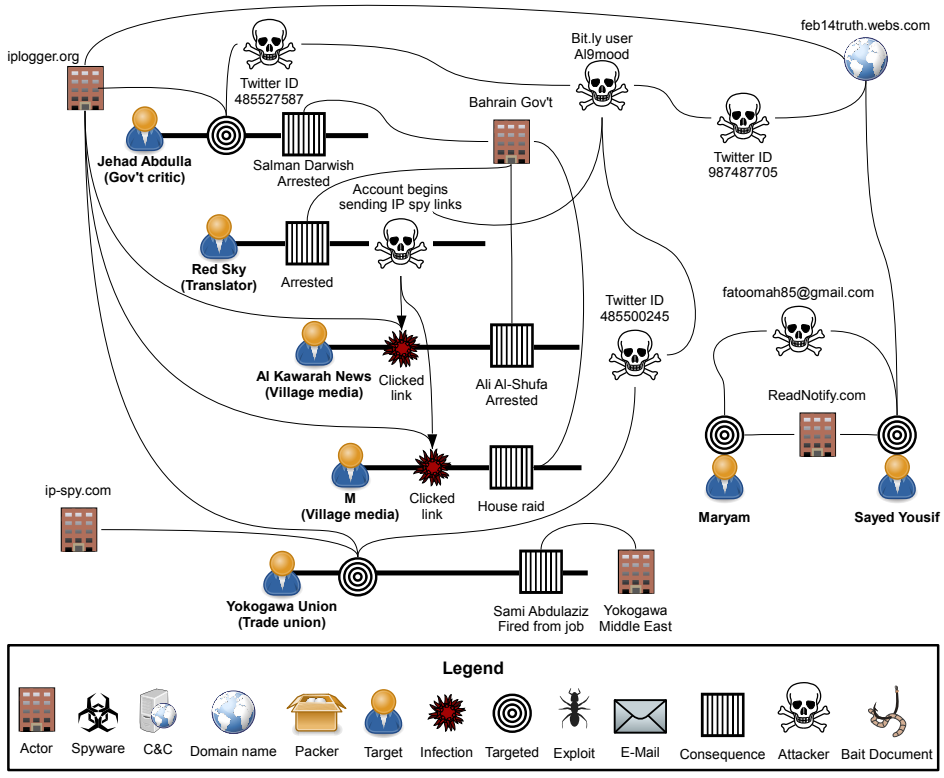


Figure 2: The ecosystem of Bahrain "IP spy" attacks.

`@alkawarahnews` (**Al Kawarah News** in Figure 2). An operator of the account forwarded us a suspicious private message sent to the Al Kawarah News Facebook account from **Red Sky**. Red Sky was purportedly arrested on 10/17/12, was convicted of insulting the King on his Twitter account `@RedSky446`, and was sentenced to four months prison.[5] When released, he found that the passwords for his Twitter, Facebook, and e-mail accounts had been changed, and did not know how to recover his accounts.

The message that Red Sky's account sent to Al Kawarah News included a link shortened using Google's `goo.gl` service. We used the `goo.gl` API to access analytics for the link, finding that it unshortened to `iplogger.org/25SX` and was created on 12/8/12. The link had received only one click, which came from Bahrain with the referrer `www.facebook.com`.

Ali's case files contained a request from the Public Prosecution for information on an IP address that it had linked to Al Kawarah News about 22 hours after the link was created. Court documents indicate that ISP data linked the IP address to Ali, and on this basis he was sentenced to one year in prison.

Red Sky also targeted **M** in Figure 2. M recalled clicking on a link from Red Sky while using an Internet connection from one of the houses in M's village. The house was raided by police on 3/12/13, who were looking for the subscriber of the house's internet connection. Police questioning revolved around Tweets that referred to Bahrain's King as a "cursed one." Red Sky had earlier targeted other users with IP spy links shortened using the *al9mood* `bit.ly` account.

The attack on **Jehad Abdulla** is noteworthy, as the account's activity aligned with communities typically critical of Bahrain's opposition. However, the account also directly criticized the King on occasion, in one case referring to him as "weak" and "stingy." An account linked to *al9mood* sent Jehad Abdulla an IP spy link on 10/2/12 in a public message. On 10/16/12, Salman Darwish was arrested for insulting the King using the Jehad Abdulla account. He was sentenced to one month in prison, partly on the basis of his confession. Salman's father claims that police denied Salman food, drink, and medical care.

Another account linked to *al9mood* targeted `@YLUBH`, the Twitter account of **Yokogawa Union**, a trade union at the Bahraini branch of a Japanese company. `@YLUBH` received at least three IP spy links in late 2012, sent via public Twitter messages. Yokogawa fired the leader of the trade union, Sami Abdulaziz Hassan, on 3/23/13 [30]. It later emerged that Sami was indeed the operator of the `@YLUBH` account, and that the police had called him in for questioning in relation to its tweets [31].

Use of embedded remote images: We identified several targets who received spoofed e-mails containing embedded remote images. Figure 2 shows two such cases, **Maryam** and **Sayed Yousif**. The attacker sent the e-mails using `ReadNotify.com`, which records the user's IP address upon

[5] According to information we received from two Twitter users, one of whom claimed to have met Red Sky in prison; another to be a colleague.

their mail client downloading the remote image.[6]

While ReadNotify.com forbids spoofing in their TOS, the service has a vulnerability known to the attackers (and which we confirmed) that allows spoofing the From address by directly setting the parameters on a submission form on their website We have not found evidence suggesting this vulnerability is publicly known, but it appears clear that the attacker exploited it, as the web form adds a X-Mailer: RNwebmail header not added when sending through ReadNotify.com's other supported methods. The header appeared in each e-mail the targets forwarded to us.

When spoofing using this method, the original sender address still appears in X-Sender and other headers. According to these, the e-mails received by the targets all came from fatoomah85@gmail.com. A link sent in one of these e-mails was connected to the *al9mood* bit.ly account.

In monitoring accounts connected to *al9mood*, we counted more than 200 IP spy links in Twitter messages and public Facebook posts. Attackers often used (1) accounts of prominent or trusted but jailed individuals like "Red Sky," (2) fake personas (e.g., attractive women or fake job seekers when targeting a labor union), or (3) impersonations of legitimate accounts. In one particularly clever tactic, attackers exploited Twitter's default font, for example substituting a lowercase "l" with an uppercase "I" or switching vowels (e.g. from "a" to an "e") to create at-a-glance identical usernames. In addition, malicious accounts tended to quickly delete IP spy tweets sent via (public) mentions, and frequently change profile names.

4.2 Syria

The use of RATs against the opposition has been a well-documented feature of the Syrian Civil War since the first reports were published in early 2012 [36, 39, 40, 32, 34]. The phenomenon is widespread, and in our experience, most members of the opposition know that some hacking is taking place. As summarized in Table 3, the attacks often include fake or maliciously packaged security tools; intriguing, or ideological, or movement-relevant content (e.g. lists of wanted persons). The seeding techniques and bait files suggest a good understanding of the opposition's needs, fears and behavior, coupled with basic familiarity with off-the-shelf RATs. In some cases attacks occur in a context that points to a more direct connection to one of the belligerents: the Syrian opposition has regularly observed that detainees' accounts begin seeding malware shortly after their arrest by government forces [41].

Researchers and security professionals have already profiled many of these RATs, including DarkComet [42, 43], Blackshades Remote Controller [38], Xtreme RAT [44], njRAT [26], and ShadowTech [36]. Some are available for purchase by anyone, in contrast to "government only" FinSpy and RCS. For example, Xtreme RAT retails for €350, while a version of Blackshades lists for €40. Others, like DarkComet, are free. We have also observed cracked versions of these RATs on Arabic-language hacker forums, making them available with little effort and no payment trail. While the RATs are cheaper and less sophisticated than FinSpy and RCS, they share the same basic functionality, including screen capture, keylogging, remote monitoring of webcams and microphones, remote shell, and file exfiltration.

In the most common attack sequence we observed, illustrated with three examples in Figure 3, the attacker seeds malware via private chat messages, posts in opposition-controlled social media groups, or e-mail. These techniques often limit the world-visibility of malicious files and links, slowing their detection by common AV products. Typically, targets receive either (1) a PE in a .zip or .rar, (2) a file download link, or (3) a link that will trigger a drive-by download. The messages usually include text, often in Arabic, that attempts to persuade the target to execute the file or click the link.

The first attacks in Figure 3 date to 2012, and use bait files with a DarkComet RAT payload. These attacks share the same C&C, 216.6.0.28, a Syrian IP address belonging to the Syrian Telecommunications Establishment, and publicly reported as a C&C of Syrian malware since February 2012 [45]. The first bait file presents to the victim as a PDF containing information about a planned uprising in Aleppo. In fact the file is a Windows Screensaver (.scr) that masquerades as a PDF using Unicode RLO, rendering a name such as ".fdp.scr" display to the victim as ".rcs.pdf." The second bait file is a dummy program containing DarkComet while masquerading as a Skype call encryption program, playing to opposition paranoia about government backdoors in common software. The third attack in Figure 3, observed in October 2013, entices targets with e-mails purporting to contain or link to videos about the current conflict, infecting victims with Xtreme RAT, and using the C&C **tn1.linkpc.net**.

For seeding, the attackers typically use compromised accounts (including those of arrested individuals) or fake identities masquerading as pro-opposition. Our illustration shows in abstract terms the use of **Victim A**'s account to seed malware ("Aleppo Plan") via (say) Skype messages to **Victim(s) B**^{n}. In the cases of **Opp. Member C** and **NGO Worker D** (here, actual victims, not abstract), targeting was by e-mail from domains apparently belonging to opposition groups, indicating a potential compromise. One domain remains active, hosting a website of the Salafist Al-Nusra front [46], while the other appears dormant. **Opp. Member C** received a malicious file as an e-mail attachment, while **NGO Worker D** was sent a shortened link (**url[.]no**/Uu5) to a download from a directory of **Mrconstrucciones[.]net**,[7] a site that may have been compromised. Both attacks resulted in an Xtreme RAT infection.

Interestingly, in the case of the fake Skype encryption the deception extended to a YouTube video from "IT Security Lab" [47] demonstrating the program's purported capabilities, as well as a website promoting the tool, **skype-encryption.sytes.net**. The attackers also constructed a basic, faux GUI for their "Encryption" program (see Figure 4). The fake GUI has a number of non-functional buttons like "Encrypt" and "DeCrypt," which generate fake prompts. While distracted by this meaningless interaction, the victim's machine is infected with DarkComet 3.3 [32, 33].

Anecdotally, campaign volume appears to track significant

[6] YahooMail and the iPhone mail client automatically load these remote images, especially in e-mails spoofed from trusted senders.

[7] Obfuscated to avoid accidental clicks on active malware URLs.

Type	Features	Examples (RATs)
Security tools	Executable files presented as a "tool" often accompanied by justifications or statements of its value in the targeted seeding, for example on a social media site, at the download location, or in videos	"Skype Encryption" (DC) [32, 33], "Facebook Security" (custom) [34], Anti-hacker (DC) [35], Fake Freegate VPN (ST) [36]
Ideologically or movement-relevant files	A document or PE as download or attachment with accompanying encouragement to open or act on the material, often masquerading as legitimate PDF documents or inadvertently leaked regime programs. Frequent use of RLO to disguise true extension (such as `.exe` or `.scr`)	"Names of individuals wanted by the Regime," (DC) "Aleppo [uprising] Plan" (DC) [37], important video (BS) [38], "Hama Rebels Council" document (DC) [39], "wanted persons" database frontend (custom), movement relevant video (njRAT), file about the Free Syrian Army (Xtreme RAT)
Miscellaneous tools	Tools pretending to offer functionality relevant to the opposition, such as a fake tool claiming to "mass report" regime pages on Facebook	hack_facebook_pro_v6.9 (DC) [40]

Table 3: Campaigns and RATs employed in Syrian surveillance. BS = Blackshades, DC = DarkComet, ST = ShadowTech.

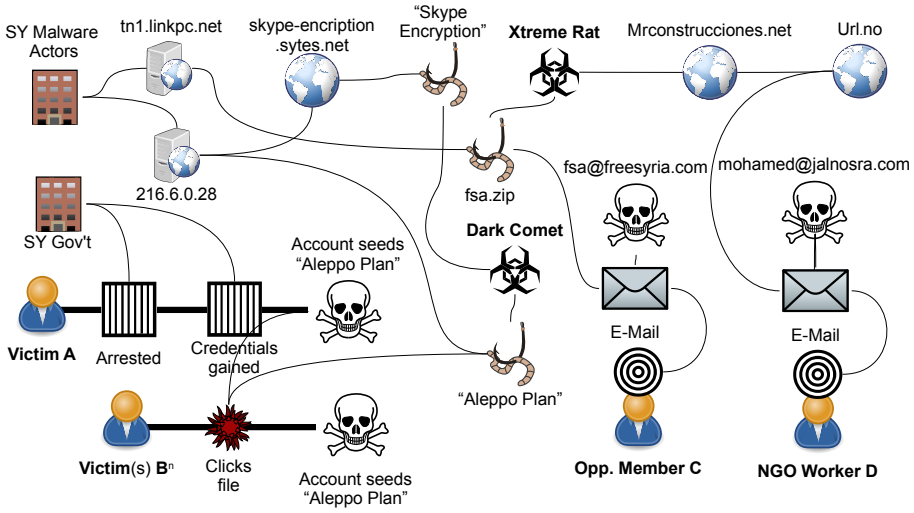


Figure 3: A sample from the ecosystem of Syrian malware campaigns.

events in the ongoing conflict. For example, campaigns dwindled and then rebounded within hours after Syria's 2012 Internet shutdown [48]. Similarly, activity observed by the authors also dwindled prior to expectation of US-led military action against Syrian government targets in September 2013. Once this option appeared to be off the table, the volume of new samples and campaigns we observed again increased, including the recent targeting of NGO workers per Figure 3. We are aware of only a negligible number of cases of the opposition using similar RATs against Syrian Government supporters, although evidence exists of other kinds of electronic attacks by third parties.

Real world consequences. The logistics and activities of Syria's numerous opposition groups are intentionally concealed from public view to protect both their efficacy, and the lives of people participating in them. Nevertheless, Syrian opposition members are generally familiar with stories off digital compromises of high-profile figures, including those entrusted with the most sensitive roles, as well as rank-and-file members. Compromise of operational security poses a documented threat to life both for victims of electronic compromise, and to family members and associates.

The Syrian conflict is ongoing, making it difficult to assem-

Figure 4: The fake Skype program distracts the victim with the promise of encrypted communications while infecting their machine with DarkComet.

ble comprehensive evidence of linkages between government actors and malware campaigns. Moreover, many individuals whose identities have been compromised are in prison or otherwise disappeared, and thus unable to relate the evidence presented to them during interrogation. Still, strong circumstantial evidence links the use of RATs, phishing, and government activity, which we briefly summarize here: (1) many Syrians have recounted to journalists and the authors how interrogators confronted them with material from their computers. For example:

> The policeman told me, "Do you remember when you were talking to your friend and you told him you had something wrong [sic] and paid a lot of money? At that time we were taking information from your laptop." [41]

(2) Syrian activists have supplied cases to international journalists [41], where arrests are quickly followed by the social media accounts of detained individuals seeding malware to contact lists (Figure 3). (3) Finally, despite the notoriety of the attack campaigns, including mention of C&C IPs in international media [45], the Syrian government has made no public statements about these campaigns nor acted to shut down the servers.

Beyond the ongoing challenges of attribution, these malware campaigns have a tangible impact on the Syrian opposition, and generally align with the interests of the Syrian government's propaganda operations. The case of Abdul Razzaq Tlass, a leader in the Free Syrian Army, is illustrative of the potential uses of such campaigns. In 2012 a string of videos emerged showing Tlass sexting and engaged in lewd activity in front of a webcam [49]. While he denied the videos, the harm to his reputation was substantial and he was eventually replaced [50].

4.3 UAE

While the UAE has experienced no recent uprising or political unrest, it has nevertheless cracked down on its opposition, concurrent with the Arab Spring.

The first attacks we observed in the UAE involved a government-grade "lawful interception" trojan known as *Remote Control System* (RCS), sold by the Italian company Hacking Team. The associated C&C server indicated direct UAE government involvement. Over time, we stopped receiving RCS samples from UAE targets, and instead observed a shift to the use of off-the-shelf RATs, and possible involvement of cyber-mercenary groups. However, poor attacker operational security allowed us to link most observed attacks together.

RCS. UAE activist **Ahmed** Mansoor (per Figure 5), imprisoned from April to November 2011 after signing an online pro-democracy petition [51], received an e-mail purportedly from "Arabic Wikileaks" in July 2012. He opened the associated attachment, "veryimportant.doc," and saw what he described as "scrambled letters". He forwarded us the e-mail for investigation.

The attachment exploited CVE-2010-3333, an RTF parsing vulnerability in Microsoft Office. The document did not contain any bait content, and part of the malformed RTF that triggered the exploit was displayed in the document. The exploit loaded shellcode that downloaded a second stage

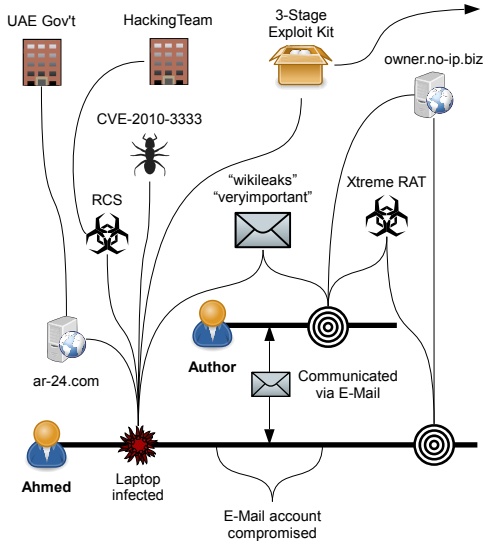


Figure 5: Part of the ecosystem of UAE surveillance attacks.

from `ar-24.com`, which in turn downloaded spyware from `ar-24.com`. We denote this combination as the **3-Stage Exploit Kit** in Figure 5.

The C&C server also ran on `ar-24.com`. When we obtained the sample in July 2012, `ar-24.com` resolved to an IP address on Linode, a hosting provider. Three months later, it resolved to a UAE address belonging to the Royal Group [52], an organization linked to the UAE government; it is chaired by Sheikh Tahnoon bin Zayed Al-Nayhan, a member of the UAE ruling family and a son of the founder of the UAE.

Identification as RCS: We identified strings in memory that matched those in a Symantec analysis [53] of RCS (also known as *DaVinci* or *Crisis*), a product of the Italian company Hacking Team [54]. We also located a structurally similar Word document via VirusTotal. The document used the same exploit and attempted to download a second stage from `rcs-demo.hackingteam.it`, which was unavailable at the time of testing.

Analysis of capabilities: RCS has a suite of functionality largely similar to FinSpy. One difference was in the vectors used to install the spyware. We located additional samples (see § 5), some of which were embedded in a `.jar` file that installs an OS-appropriate version of RCS (Windows or OSX), optionally using an exploit. If embedded as an applet, and no exploit is present, Java displays a security warning and asks the user whether they authorize the installation. We also saw instances of the **3-Stage Exploit Kit** where the first stage contained a Flash exploit; in some cases, we could obtain all stages and confirm that these installed RCS. Some samples were packed with the MPress packer [55], and some Windows samples were obfuscated to look like the `PuTTY` SSH client.

Another difference is in persistence. For example, the RCS sample sent to Ahmed adds a `Run` registry key, whereas the FinSpy samples used in Bahrain overwrite the hard disk's boot sector to modify the boot process; the spyware is loaded be-

fore the OS, and injects itself into OS processes as they start. The RCS samples we examined also had the ability to propagate to other devices, including into inactive VMWare virtual machines by modifying the disk image, onto USB flash drives, and onto Windows Mobile phones. We did not observe similar capabilities in the FinSpy samples we examined.

Exploitation of captured data: When Ahmed Mansoor received the RCS document, he opened it, infecting his computer (Figure 5). Ahmed subsequently noted several suspicious accesses to his GMail account using IMAP. Even after he changed his password, the accesses continued. While corresponding with Ahmed on his compromised account, an author of this paper discovered that the attackers had installed an *application-specific password* [56] in Ahmed's GMail account, a secondary password that they apparently used to access his account even after he changed his main password. The suspicious accesses stopped after removal of the application-specific password.

Two weeks after this correspondence with Ahmed, one of us (**Author** in Figure 5) received a targeted e-mail with a link to a file hosted on Google Docs containing a commercial off-the-shelf RAT, Xtreme RAT. The e-mail was sent from the UAE's timezone (as well as of other countries) and contained the terms "veryimportant" and "wikileaks", just like in the e-mail received by Ahmed.

The instance of Xtreme RAT sent to **Author** used `owner.no-ip.biz` for its C&C, one of the domains mentioned in a report published by Norman about a year-long campaign of cyberattacks on Israeli and Palestinian targets carried out by a group that Norman was unable to identify [57]. Three months after **Author** was targeted, Ahmed received an e-mail containing an attachment with Xtreme RAT that talked to the same C&C server (Figure 5), suggesting that the attackers who infected Ahmed with RCS may have provided a list of interesting e-mail addresses to another group for further targeting.

Possible consequences: Shortly after he was targeted, Ahmed says he was physically assaulted twice by an attacker who appeared able to track Ahmed's location [58]. He also reports that his car was stolen, a large sum of money disappeared from his bank account, and his passport was confiscated [59]. He believes these consequences are part of a government intimidation campaign against him, but we did not uncover any direct links to his infection. (Interestingly, spyware subsequently sent to others has used bait content about Ahmed.)

Further attacks: In October 2012, UAE **Journalist A** and **H**uman **R**ights **activist B** (per Figure 6) forwarded us suspicious e-mails they had received that contained a Word document corresponding to the first stage of **3-Stage Exploit Kit** (Figure 5). The attachment contained an embedded Flash file that exploited a vulnerability fixed in Adobe Flash 11.4, loading shell code to download a second stage from `faddeha.com`. We were unable to obtain the second stage or the ultimate payload, as the website was unavailable at the time of testing. However, the exploit kit appears indicative of Hacking Team involvement. A page on `faddeha.com` found in Google's cache contained an embedded `.jar` with the same applet class (*WebEnhancer*) as those observed in other `.jar` files that we found to contain RCS.

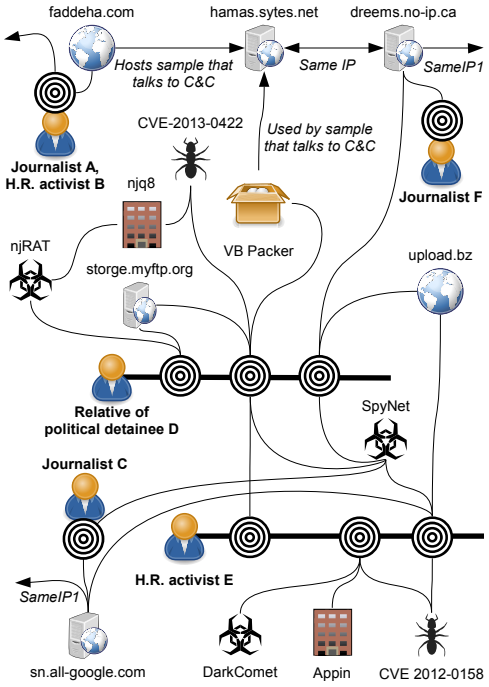


Figure 6: Another part of the ecosystem of UAE surveillance attacks.

Off-the-shelf RATs. We found a file that VirusTotal had downloaded from `faddeha.com`, which appeared to be a remote access toolkit known as *SpyNet*, available for general purchase for 50 Euros [60]. The SpyNet sample communicated with the C&C `hamas.sytes.net`.

SpyNet Packing: We found another instance of the first stage of the **3-Stage Exploit Kit** on VirusTotal. The exploit downloaded a second stage, which in turn downloaded a sample of SpyNet from `maile-s.com`. This sample of SpyNet communicated with the same C&C `hamas.sytes.net`. The sample was packed using *ASProtect* [61]. When run, the sample unpacks a compiled Visual Basic project that loads, via the `RunPE` method [62], an executable packed with *UPX* [63]. Finally, this executable unpacks SpyNet. SpyNet's GUI only offers an option to pack with UPX, suggesting that the attackers specially added the other layers of packing. In some cases, the Visual Basic project bears the name *NoWayTech*, which appears to be an underground `RunPE` tool, while others are named *SpyVisual*, which we have been unable to trace to any public underground tools, and thus also may reflect customization by the attacker. The *SpyVisual* projects contain the string `c:\Users\Zain\AppData\Local\Temp\OLE1EmbedStrm.wav`, which we used as the fingerprint **VB Packer** in Figure 6.

Cedar Key attack: The same **VB Packer** was used in an attack on **Relative of political detainee D** and **H.R. activist E** in Figure 6. These individuals received e-mails containing a link to a web page hosted on `cedarkeyrv.com` impersonating YouTube. Loading the page greeted the target with "*Video loading please wait ...*" The page redirected to a YouTube video a few seconds later, but first loaded a Java exploit [64]—a

known vulnerability with no patch at the time that the e-mails were sent. Oracle released a patch 12 hours after activists began receiving these links.

The cedarkeyrv.com domain is associated with an RV park in Cedar Key, Florida. The website's hosting company told us that the site had apparently suffered a compromise, but did not have further details.

The exploit used in the attack appears to have been originally posted by a Kuwaiti user, *njq8*, on an Arabic-language exploit sharing site [65]. We contacted *njq8*, who told us that he had obtained the exploit elsewhere and modified it prior to posting. The attack downloaded an instance of SpyNet from isteeler.com (which from our inspection did not appear to have any legitimate content), which used the C&C storge.myftp.org. This same C&C occurred in another attack (Figure 6) targeting **Relative of political detainee D**; in that case, the payload was a freely-available RAT known as *njRAT*, written by the same *njq8* as the exploit-poster discussed above. However, we did not find any other evidence suggesting *njq8*'s involvement in either attack.

More SpyNet attacks: The domain hamas.sytes.net, which we previously saw used by two SpyNet samples, resolved to 67.205.79.177. Historically, dreems.no-ip.ca also resolved to this address. An unidentified dropper using this C&C targeted **Journalist F**; a SpyNet attack on **Relative of political detainee D** also used this C&C. In that latter case, the sample arrived via e-mail in a .rar attachment that contained an .scr file disguised as a Word document. The .scr file was a self-extracting archive that decompressed and ran both the bait document and the payload. The SMTP source of the e-mail was webmail.upload.bz.

Appin: In early 2013 UAE **H.R. activist E** forwarded numerous documents that included a particular CVE-2012-0158 exploit for Microsoft Word. In all, these totaled 17 distinct hashes of documents, and 10 distinct hashes of payloads (some documents that differed in their hash downloaded the same payload). The exploits primarily downloaded instances of SpyNet from upload.bz, which for the most part communicated with C&C at sn.all-google.com. This domain was also used for C&C in other attacks, including that on **Journalist C**.

Two of the other CVE-2012-0158 exploits downloaded DarkComet from www.getmedia.us and www.technopenta.com after posting system information to random123.site11.com. All three domains match those used by an Indian cybermercenary group said to be linked to Appin Security Group [66]. The former two domains hosted content other than spyware (i.e., they may have been compromised). We alerted the owner of www.getmedia.us, who removed the payloads.

5 Empirical characterization

The samples we received afforded us an opportunity to empirically characterize the use of FinFisher and Hacking Team around the world, enabling us to assess their prevalence, and identify other country cases that may warrant future investigation. We analyzed the samples and the behavior of their C&C servers to develop indicators (fingerprints) for how the servers respond to certain types of requests. We then scanned the full Internet IPv4 address space ("/0") for these, along with probing results found by past scans. In many cases we do not release the full details of our fingerprints to avoid compromising what may be legitimate investigations.

5.1 FinSpy

Identifying and linking servers: We developed a number of fingerprints for identifying FinSpy servers using HTTP-based probing as well as FinSpy's custom TLV-based protocol. We leveraged quirks such as specific non-compliance with RFC 2616, responses to certain types of invalid data, and the presence of signatures such as the bizarre "Hallo Steffi" that Guarnieri identified from Bahraini FinSpy C&C servers [67, 68]. See Appendix A for details. We then exhaustively scanned the Internet looking for matches to these fingerprints.

Gamma documentation advertises that an operator of FinSpy can obscure the location of the C&C server (called the *master*) by setting up a proxy known as a *relay*. In Spring 2013 we noticed FinSpy servers now issuing 302 Redirects to google.com. However, we noticed anomalies: for example, servers in India were redirecting to the Latvian version of Google google.lv. We suspect that the server in India was a relay forwarding to a master in Latvia. Because the master served as a proxy for Google, we could uncover its IP address using a Google feature that prints a user's IP address for the query "IP address." We created an additional fingerprint based on the proxying behavior and issued GET /search?q=ip+address&nord=1 requests to servers We note some interesting master locations in Table 4.

Server locations: In all, our fingerprints matched 92 distinct IP addresses in 35 different countries. Probing these on 8/8/13 revealed 22 distinct addresses still responding, sited in: Bahrain, Bangladesh, Bosnia and Herzegovina, Estonia, Ethiopia, Germany, Hong Kong, Indonesia, Macedonia, Mexico, Romania, Serbia, Turkmenistan, and the United States. We found servers responding to a number of our fingerprints, suggesting either that some servers lag in their updates, or a concerted effort to vary the behavior of FinSpy servers to make detection harder.

We found: **(1)** 3 IP addresses in ranges registered to Gamma. **(2)** Servers in 3 IP ranges explicitly registered to government agencies: Turkmenistan's Ministry of Communications, Qatar's State Security Bureau, and the Bulgarian Council of Ministers. **(3)** 3 additional IP addresses in Bahrain, all in Batelco. **(4)** Servers in 7 countries with governments classified as "authoritarian regimes" by *The Economist* [69]: Bahrain, Ethiopia, Nigeria, Qatar, Turkmenistan, UAE, Vietnam.

Additional FinSpy samples: In parallel to our scanning, we obtained 9 samples of FinSpy by writing YARA [70] rules for the "malware hunting" feature of VirusTotal Intelligence. This feature sends us all newly-submitted samples that match our signatures. We located a version of FinSpy that does not use the normal FinSpy handshake, but instead uses a protocol based on HTTP POST requests for communication with the C&C server. This did not appear to be an older or newer ver-

Relay IP	Relay Block Assignment	Relay Country	Master IP	Master Block Assignment	Master Country
5.199.xxx.xxx	SynWebHost	Lithuania	188.219.xxx.xx	Vodafone	Italy
46.23.xxx.xxx	UK2 VPS.net	UK	78.100.xxx.xxx	State Security Building	Qatar
119.18.xxx.xxx	HostGator	India	81.198.xxx.xxx	Statoil DSL	Latvia
180.235.xxx.xxx	Asia Web Services	Hong Kong	80.95.xxx.xxx	T-Systems	Czech Republic
182.54.xxx.xxx	GPLHost	Australia	180.250.xxx.xxx	PT Telekom	Indonesia
206.190.xxx.xxx	WestHost	USA	112.78.xxx.xxx	Biznet ISP	Indonesia
206.190.xxx.xxx	Softlayer	USA	197.156.xxx.xxx	Ethio Telecom	Ethiopia
209.59.xxx.xxx	Endurance International	USA	59.167.xxx.xxx	Internode	Australia
209.59.xxx.xxx	Endurance International	USA	212.166.xxx.xxx	Vodafone	Spain

Table 4: Deproxifying FinSpy (mapping initial C&C IP addresses to the masters to which they forward).

sion of the protocol, suggesting that our scan results may not reveal the full scope of FinSpy C&C servers. Perhaps, the `HTTP POST` protocol was only delivered to a specific Gamma customer to meet a requirement.

5.2 Remote Control System (RCS)

We began by analyzing the UAE RCS sample from Ahmed and 6 samples obtained from VirusTotal by searching for AV results containing the strings "DaVinci" and "RCS." At the time, several AV vendors had added detection for RCS based on a sample analyzed by Dr. Web [71] and the UAE RCS sample sent to Ahmed. We also similarly obtained and analyzed samples of FSBSpy [72], a piece of malware that can report system information, upload screenshots, and drop and execute more malware, Based on these samples, we devised YARA signatures that yielded 23 additional samples of structurally similar malware.

Fingerprints: We probed the C&C servers of the RCS and FSBSpy samples, and found that they responded in a distinctive way to HTTP requests, and returned distinctive SSL certificates.

We searched sources including Shodan, 5 Internet Census service probes [73], and Critical.IO scanning data [68] for the observed distinctive HTTP behavior. We searched for the distinctive SSL certificates in two Internet Census service probes, and SSL certificate scans from ZMap [74]. We also contacted a team at TU Munich [75], who applied our fingerprints to their SSL scanning data. Across all of these sources, we obtained 31,345 indicator hits reflecting 555 IP addresses in 48 countries.

One SSL certificate returned by 175 of the servers was issued by "/CN=RCS Certification Authority /O=HT srl," apparently referring to the name of the spyware and the company. Servers for 5 of our FSBSpy samples and 2 of our RCS samples responded with this type of certificate.

Some servers returned these certificates in chains that included another distinctive certificate. We found 175 distinct IP addresses (including the C&C's for 5 of our FSBSpy samples and 2 of our RCS samples) responded with this second type of certificate.

We devised two more indicators: one that matched 125 IP addresses, including 7 of our FSBSpy samples' C&C's, and one that matched 2 IP addresses, in Italy and Kazakhstan.

Server locations: On 11/4/13 we probed all of the IP addresses that we collected, finding 166 active addresses matching one of our fingerprints in 29 different countries. We summarize the top providers and countries in Table 5.

Country	IPs
United States	61
United Kingdom	18
Italy	16
Japan	10
Morocco	7

Provider	IPs
Linode	42
NOC4Hosts	16
Telecom Italia	9
Maroc Telecom	7
InfoLink	6

Table 5: Top countries and hosting providers for RCS servers active on 11/4/13.

The prevalence of active servers either located in the USA or hosted by Linode is striking,[8] and seems to indicate a pervasive use of out-of-country web hosting and VPS services.

In addition, we found: **(1)** 3 IP addresses on a `/28` named "HT public subnet" that is registered to the CFO of Hacking Team [76]. The domain `hackingteam.it` resolves to an address in this range. **(2)** An address belonging to Omantel, a majority-state-owned telecom in Oman. This address was unreachable when we probed it; a researcher pointed us to an FSBSpy sample that contained an Arabic-language bait document about Omani poetry, which talked to a C&C in the UK. **(3)** 7 IP addresses belonging to Maroc Telecom. Moroccan journalists at Mamfakinch.com were previously targeted by RCS in 2012 [77]. **(4)** Overall, servers in 8 countries with governments deemed "authoritarian regimes" [69]: Azerbaijan, Kazakhstan, Nigeria, Oman, Saudi Arabia, Sudan, UAE, Uzbekistan.

Link to Hacking Team: All active servers matching one of our signatures also responded peculiarly when queried with particular ill-formed HTTP requests, responding with "`HTTP1/1 400 Bad request`" (should be "`HTTP/1.1`") and a body of "`Detected error: HTTP code 400`". Googling for this response yielded a GitHub project `em-http-server` [78], a Ruby-based webserver. The project's author is listed as Alberto Ornaghi, a software architect at Hacking Team. We suspect that the Hacking Team C&C server code may incorporate code from this project.

Links between servers: We identified many cases where several servers hosted by different providers, and in different countries, returned identical SSL certificates matching our fingerprints. We also observed 30 active servers used a global IPID. Only one active server had neither a global IPID nor

[8] 19 of the 42 Linode servers were hosted in the USA, so the two patterns of prevalence are mostly distinct.

an SSL certificate matching our fingerprints. We assessed whether servers returning SSL certificates were forwarding to the servers with global IPIDs by inducing bursts of traffic at the former and monitoring the IPID at the latter. For 11 servers, we found that the server's activity correlated to bursts sent to other servers We grouped servers by the SSL certificates they returned, and found that each group forwarded to only a single server, except for one case where a group forwarded to two different IPs (both in Morocco). We also found two groups that forwarded to the same address. There was a 1:1 mapping between the remaining 8 addresses and groups. We refer to a group along with the server(s) it forwards to as a *server group*. We identified several server groups that may be associated with victims or operators in a certain country. Some of these suggest possible further investigation:

Turkey: We identified a group containing 20 servers in 9 countries. Two RCS and 5 FSBSpy samples from VirusTotal communicated with various servers in the group. The RCS samples also communicated with domains with lapsed registrations, so we registered them to observe incoming traffic. We exclusively received RCS traffic from Turkish IP addresses. (RCS traffic is identifiable based on a distinctive user agent and URL in `POST` requests.) A sample of FSBSpy apparently installed from an exploit on a Turkish server talked to one of the servers in this group.[79]

We also found server groups containing servers in **Uzbekistan** and **Kazakhstan**; we found FSBSpy samples on VirusTotal uploaded from these countries that communicated with servers in these groups.

In the above cases, save Turkey, the country we have identified is classified as an "authoritarian regime," and may be using Hacking Team products against the types of targets we profile in this paper. In the case of Turkey, there are hints that the tool may be employed against dissidents [80].

6 Summary

Targeted surveillance of individuals conducted by nation-states poses an exceptionally challenging security problem, given the great imbalance of resources and expertise between the victims and the attackers. We have sketched the nature of this problem space as reported to us by targeted individuals in three Middle Eastern countries. The attacks include spyware for ongoing monitoring and the use of "IP spy" links to deanonymize those who voice dissent.

The attacks, while sometimes incorporating effective social engineering, in general lack novel technical elements. Instead, they employ prepackaged tools developed by vendors or acquired from the cybercrime underground. This technology sometimes suffers from what strike us as amateurish mistakes (multiple serious errors implementing cryptography, broken protocol messages), as does the attackers' employment of it (identifying-information embedded in binaries, C&C servers discoverable via scanning or "Google hacking", clusters of attack accounts tied by common activity). Some of these errors assisted our efforts to assemble strong circumstantial evidence of governmental origins. In addition, we mapped out the global use of two "governmental" hacking suites, including identifying 11 cases in which they appeared to be used in countries governed by "authoritarian regimes."

We aim with this work to inspire additional research efforts addressing the difficult problem of how to adequately protect individuals with very limited resources facing very powerful adversaries. Open questions include robust, practical detection of targeted attacks designed to exfiltrate data from a victim's computer, as well as detection of and defense against novel attack vectors, like tampering with Internet connections to insert malware.

The task is highly challenging, but the potential stakes are likewise very high. An opposition member, reflecting on government hacking in Libya, speculated as to why some users would execute files even while recognizing them as potentially malicious [2]: "*If we were vulnerable we couldn't care less … we were desperate to get our voices out … it was a matter of life or death … it was just vital to get this information out.*"

Acknowledgment

This work was supported by the National Science Foundation under grants 1223717 and 1237265, and by a Citizen Lab Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

The authors would like to thank the following individuals for their help in various aspects of our analysis: Bernhard Amman, Collin D. Anderson, Brandon Dixon, Zakir Durumeric, Eva Galperin, Claudio Guarnieri, Drew Hintz, Ralph Holz, Shane Huntley, Andrew Lyons, Mark Schloesser, and Nicholas Weaver.

References

[1] "Dark Secrets—Hacking Team commercial," accessed: 12-November-2013. [Online]. Available: http://bit.ly/1bCh57v

[2] J. Scott-Railton, "Revolutionary Risks: Cyber Technology and Threats in the 2011 Libyan Revolution," US Naval War College, Tech. Rep., 2013.

[3] S. H. AlJalahma, "Response to The Guardian—UK companys software used against Bahrain activist," May 2013, accessed: 12-November-2013. [Online]. Available: http://bit.ly/19iVUUP

[4] V. Silver, "Gamma Says No Spyware Sold to Bahrain; May Be Stolen Copy," Jul. 2012, accessed: 12-November-2013. [Online]. Available: http://bloom.bg/17SOXQs

[5] A. Jeffries, "Meet Hacking Team, the company that helps the police hack you," Sep. 2013, accessed: 12-November-2013. [Online]. Available: http://bit.ly/1bCajyl

[6] T. Brewster, "From Bahrain To Belarus: Attack Of The Fake Activists," Jul. 2013, accessed: 12-November-2013. [Online]. Available: http://bit.ly/1gIgwhW

[7] V. Silver, "MJM as Personified Evil Says Spyware Saves Lives Not Kills Them," 2011, accessed: 12-November-2013. [Online]. Available: http://bloom.bg/170E8sQ

[8] D. Gilbert, "Hacking Team and the Murky World of State-Sponsored Spying," 2013, accessed: 12-November-2013. [Online]. Available: http://bit.ly/17tBBtm

[9] R. Clayton, S. J. Murdoch, and R. N. Watson, "Ignoring the Great Firewall of China," in *PETS*. Springer, 2006, pp. 20–35.

[10] J. R. Crandall et al., "ConceptDoppler: A Weather Tracker for Internet Censorship," in *ACM CCS*, 2007.

[11] X. Xu, Z. M. Mao, and J. A. Halderman, "Internet Censorship in China: Where Does the Filtering Occur?" in *Proc. PAM*, 2011.

[12] M. Sherr, G. Shah, E. Cronin, S. Clark, and M. Blaze, "Can They Hear Me Now? A Security Analysis of Law Enforcement Wiretaps," in *ACM CCS*, 2009, pp. 512–523.

[13] J. R. Crandall, M. Crete-Nishihata, and J. Knockel, "Chat program censorship and surveillance in China: Tracking TOM-Skype and Sina UC," *First Monday*, vol. 18, no. 7, Jul. 2013, accessed: 8-August-2013. [Online]. Available: http://bit.ly/1fzNcHl

[14] S. Wolchok, R. Yao, and J. A. Halderman, "Analysis of the Green Dam Censorware System," Tech. Rep., 2009.

[15] F. Li, A. Lai, and D. Ddl, "Evidence of Advanced Persistent Threat: A case study of malware for political espionage," in *MALWARE*, 2011.

[16] "Default https access for Gmail," 2010, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1bBktPM

[17] "Making Twitter more secure: HTTPS," 2011, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1i7l9kM

[18] L. Constantin, "Facebook to roll out HTTPS by default to all users," 2012, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1bsLBCm

[19] "FinFisher: Governmental IT Intrusion and Remote Monitoring Solutions," accessed: 12-November-2013. [Online]. Available: http://bit.ly/1840Lxn

[20] "BlackBerry rogue software leaves sour taste in UAE," 2013, accessed: 11-November-2013. [Online]. Available: http://on.ft.com/HVXvJP

[21] Mandiant, "The Advanced Persistent Threat," 2010.

[22] ——, "APT1: Exposing One of China's Cyber Espionage Units," 2013.

[23] S. Fagerland, M. Krakvik, J. Camp, and N. Moran, "Operation Hangover: Unveiling an Indian Cyberattack Infrastructure," 2013.

[24] R. Deibert and R. Rohozinski, "Tracking GhostNet: Investigating a Cyber Espionage Network," *Information Warfare Monitor*, p. 6, 2009.

[25] S. Nagaraja and R. Anderson, "The snooping dragon: social-malware surveillance of the Tibetan movement," Tech. Rep., 2009.

[26] F. C. Solutions, ""njRAT" Uncovered," 2013, accessed: 25-June-2013. [Online]. Available: http://bit.ly/1eJheel

[27] "FinFisher - Excellence in IT Investigation," accessed: 27-February-2014. [Online]. Available: http://www.finfisher.com/

[28] R. Rolles, "Unpacking virtualization obfuscators," in *USENIX WOOT*, 2009.

[29] "TEMU: The BitBlaze Dynamic Analysis Component," accessed: 7-August-2013. [Online]. Available: http://bit.ly/1clcxSZ

[30] "'Reinstate sacked official' call," 2013, accessed: 11-November-2013. [Online]. Available: http://bit.ly/1aRUZ4b

[31] "Unionist Questioned," 2013, accessed: 23-April-2013. [Online]. Available: http://bit.ly/1gHnBiS

[32] N. Villeneuve, "Fake Skype Encryption Service Cloaks DarkComet Trojan," Apr. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/17SpA1c

[33] E. Galperin and M. Marquis-Boire, "Fake YouTube Site Targets Syrian Activists With Malware," Mar. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/HSCRet

[34] ——, "New Wave of Facebook Phishing Attacks Targets Syrian Activists," Apr. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/1hDQsG8

[35] ——, "Pro-Syrian Government Hackers Target Activists With Fake Anti-Hacking Tool," Aug. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/1eJj12T

[36] J. Scott-Railton and M. Marquis-Boire, "A Call to Harm: New Malware Attacks Target the Syrian Opposition," Citizen Lab, Tech. Rep., Jun. 2013, accessed: 3-August-2013. [Online]. Available: http://bit.ly/1a2l9PK

[37] E. Galperin and M. Marquis-Boire, "Trojan Hidden in Fake Revolutionary Documents Targets Syrian Activists," May 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/1cSJT0

[38] M. Marquis-Boire and S. Hardy, "Syrian Activists Targeted with BlackShades Spy Software," Jun. 2012, accessed: 12-November-2013. [Online]. Available: http://bit.ly/1a2l6mX

[39] S. Fagerland, "The Syrian Spyware," Feb. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/HLyGR9

[40] Telecomix, "REPORT of a Syrian spyware," p. 9, Feb. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/1bsNcIk

[41] S. Faris, "The Hackers of Damascus," Nov. 2012, accessed: 9-August-2013. [Online]. Available: http://buswk.co/17t8RRH

[42] L. Aylward, "Malware Analysis—Dark Comet RAT," Nov. 2011, accessed: 4-August-2013. [Online]. Available: http://bit.ly/16ZXgag

[43] Quequero, "DarkComet Analysis—Understanding the Trojan used in Syrian Uprising," Mar. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/19i6kEl

[44] S. Denbow and J. Hertz, "Pest Control: Taming the RATs," p. 14, accessed: 12-November-2013. [Online]. Available: http://bit.ly/1fzLA0m

[45] B. Brumfield, "Computer spyware is newest weapon in Syrian conflict," Feb. 2012, accessed: 4-August-2013. [Online]. Available: http://cnn.it/HLz5TA

[46] "jalnosra.com," accessed: 27-February-2014. [Online]. Available: jalnosra.com

[47] "Skype Encryption.wmv," accessed: 27-February-2014. [Online]. Available: http://bit.ly/HZ3e1y

[48] E. Galperin and M. Marquis-Boire, "The Internet is Back in Syria and So is Malware Targeting Syrian Activists," Dec. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/1bngqFc

[49] "Free Syrian Army Sex Tape—Abdul Razzaq Tlass [NSFW]," accessed: 5-August-2013. [Online]. Available: http://bit.ly/1gHqDDH

[50] A. Lund, "Holy Warriors: A field guide to Syria's jihadi groups," Oct. 2012, accessed: 5-August-2013. [Online]. Available: http://atfp.co/17t8yq5

[51] "Ahmed Mansoor and Four Other Pro-Democracy Activists Pardoned and Freed," 2013, accessed: 10-November-2013. [Online]. Available: http://bit.ly/18pHpis

[52] "Royal Group," accessed: 27-February-2014. [Online]. Available: http://www.royalgroupuae.com/

[53] T. Katsuki, "Crisis for Windows Sneaks onto Virtual Machines," 2012, accessed: 27-February-2014. [Online]. Available: http://bit.ly/MzheRJ

[54] "Hacking Team," accessed: 27-February-2014. [Online]. Available: http://www.hackingteam.it/

[55] "MPRESS," accessed: 27-February-2014. [Online]. Available: http://www.matcode.com/mpress.htm

[56] "Sign in using application-specific passwords," accessed: 27-February-2014. [Online]. Available: https://support.google.com/accounts/answer/185833?hl=en

[57] S. Fagerland, "Systematic cyber attacks against Israeli and Palestinian targets going on for a year," 2012, accessed: 12-November-2013. [Online]. Available: http://bit.ly/1aSdw07

[58] V. Silver, "Spyware Leaves Trail to Beaten Activist Through Microsoft Flaw," 2012, accessed: 14-November-2013. [Online]. Available: http://bloom.bg/1ja2geI

[59] B. Hubbard, "Emirates Balk at Activism in Region Hit by Uprisings," 2013, accessed: 14-November-2013. [Online]. Available: http://nyti.ms/I4n2Aw

[60] "SPY NET," accessed: 27-February-2014. [Online]. Available: http://newspynetrat.blogspot.com/

[61] "Asprotect SKE," accessed: 27-February-2014. [Online]. Available: http://www.aspack.com/asprotect32.html

[62] "Unpacking VBInject/VBCrypt/RunPE," 2010, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1e28nS2

[63] "Ultimate Packer for eXecutables," accessed: 27-February-2014. [Online]. Available: http://upx.sourceforge.net/

[64] "CVE-2013-0422," accessed: 27-February-2014. [Online]. Available: http://bit.ly/NA1O0A

[65] njq8, "New java drive-by 2013-1-11," 2013, accessed: 27-February-2014. [Online]. Available: http://www.devpoint.com/vb/t357796.html

[66] "Appin Technology Lab," accessed: 27-February-2014. [Online]. Available: http://www.appinonline.com/

[67] C. Guarnieri, "Analysis of the FinFisher Lawful Interception Malware," 2012, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1eJjVMV

[68] H. Moore, "Critical Research: Internet Security Survey," 2012.

[69] "Democracy Index 2012: Democracy at a Standstill," 2012, accessed: 7-August-2013. [Online]. Available: http://bit.ly/HSEDMD

[70] "YARA - The pattern matching swiss knife for malware researchers," accessed: 27-February-2014. [Online]. Available: http://plusvic.github.io/yara/

[71] "Cross-platform Trojan controls Windows and Mac machines," 2012, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1eJnJgZ

[72] S. Golovanov, "Adobe Flash Player 0-day and HackingTeam's Remote Control System," 2013, accessed: 7-August-2013. [Online]. Available: http://bit.ly/17n12ro

[73] "Internet Census 2012," 2013, accessed: 7-August-2013. [Online]. Available: http://bit.ly/1i7rRHs

[74] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-Wide Scanning and its Security Applications," in *USENIX Security*, Aug. 2013.

[75] "Home of Crossbear and OONIBear," accessed: 27-February-2014. [Online]. Available: https://pki.net.in.tum.de/

[76] "RIPE Database Query for FASTWEB-HT," accessed: 27-February-2014. [Online]. Available: http://bit.ly/MzkigV

[77] "How Government-Grade Spy Tech Used A Fake Scandal To Dupe Journalists," 2012, accessed: 7-August-2013.

[78] A. Ornaghi, "em-http-server," accessed: 27-February-2014. [Online]. Available: https://github.com/alor/em-http-server

[79] SophosLabs, "Anatomy of a targeted attack—SophosLabs explores an Adobe zero-day "malware experiment"," 2013, accessed 7-August-2013. [Online]. Available: http://bit.ly/HQ1oRc

[80] K. Zetter, "American Gets Targeted by Digital Spy Tool Sold to Foreign Governments," 2013, accessed: 14-November-2013. [Online]. Available: http://wrd.cm/1fHonth

[81] M. Marquis-Boire and B. Marczak, "From Bahrain With Love: FinFisher's Spy Kit Exposed?" Jul. 2012, accessed: 4-August-2013. [Online]. Available: http://bit.ly/1bngpB2

A FinSpy fingerprints

Previous work by Guarnieri on scanning for FinSpy servers found that in response to a request such as `GET /`, the Bahraini FinSpy C&C server returns a response with the string "`Hallo Steffi`" [67]. Guarnieri searched a database of such responses compiled by the Critical.IO Internet scanning project [68], locating 11 additional servers in 10 countries [67]. We refer to this fingerprint as α_1. Concurrent with this effort, we devised our own fingerprint β_1 that tested three aspects of the handshake between a FinSpy infectee and a FinSpy C&C server, which follows a custom TLV-based protocol running on ports such as 22, 53, 80, and 443. We conducted targeted scanning of several countries using β_1, and also confirmed Guarnieri's findings for those servers still reachable after he published his findings.

We observed a trend: changes in HTTP response behavior by FinFisher after publication of findings about the software. In July 2012, for example, after a post about Bahraini FinSpy samples [81], servers closed the TCP connection in response to a `GET /` or `HEAD /` request (although servers continued to behave consistently with β_1. Other changes followed later in 2012, including a new response to `GET /` requests that included an imperfect copy of an Apache server's HTTP response (the `Date` header used `UTC` rather than `GMT`). We fingerprinted this error as α_2, and later in 2012 fingerprinted other distinctive behavior in response to `GET /` requests as α_3.

Subsequent scans of `/0` for α_2 and α_3, and five service probes of the Internet Census for α_1 through α_3, located several additional servers. In Feburary 2013 we identified and fingerprinted new HTTP response behavior with α_4 and modified β_1 to produce β_2, which tests only two of the three aspects of the FinSpy handshake (the third test of β_1 was broken when FinSpy servers were updated to accept types of invalid data they had previously rejected).

As of 3/13/13, all servers that matched any α fingerprint matched β_2.

Targeted Threat Index: Characterizing and Quantifying Politically-Motivated Targeted Malware

Seth Hardy[§] Masashi Crete-Nishihata[§] Katharine Kleemola[§] Adam Senft[§]
Byron Sonne[§] Greg Wiseman[§] Phillipa Gill[†] Ronald J. Deibert[§]
§ *The Citizen Lab, Munk School of Global Affairs, University of Toronto, Canada*
† *Stony Brook University, Stony Brook, USA*

Abstract

Targeted attacks on civil society and non-governmental organizations have gone underreported despite the fact that these organizations have been shown to be frequent targets of these attacks. In this paper, we shed light on targeted malware attacks faced by these organizations by studying malicious e-mails received by 10 civil society organizations (the majority of which are from groups related to China and Tibet issues) over a period of 4 years.

Our study highlights important properties of malware threats faced by these organizations with implications on how these organizations defend themselves and how we quantify these threats. We find that the technical sophistication of malware we observe is fairly low, with more effort placed on socially engineering the e-mail content. Based on this observation, we develop the Targeted Threat Index (TTI), a metric which incorporates both social engineering and technical sophistication when assessing the risk of malware threats. We demonstrate that this metric is more effective than simple technical sophistication for identifying malware threats with the highest potential to successfully compromise victims. We also discuss how education efforts focused on changing user behaviour can help prevent compromise. For two of the three Tibetan groups in our study simple steps such as avoiding the use of email attachments could cut document-based malware threats delivered through e-mail that we observed by up to 95%.

1 Introduction

Civil society organizations (CSOs), working on human rights issues around the globe, face a spectrum of politically-motivated information security threats that seek to deny (e.g. Internet filtering, denial-of-service attacks), manipulate (e.g. website defacements) or monitor (e.g. targeted malware) information related to their work. Targeted malware attacks in particular are an increasing problem for CSOs. These attacks are not isolated incidents, but waves of attacks organized in campaigns that persistently attempt to compromise systems and gain access to networks over long periods of time while remaining undetected. These campaigns are custom designed for specific targets and are conducted by highly motivated attackers. The objective of these campaigns is to extract information from compromised systems and monitor user activity and is best understood as a form of espionage. CSOs can be particularly susceptible to these threats due to limited resources and lack of security awareness. Targeted malware is an active research area, particularly in private industry. However, focused studies on targeted attacks against CSOs are relatively limited despite the persistent threats they face and the vulnerability of these groups.

In this study, we work with 10 CSOs for a period of 4 years to characterize and track targeted malware campaigns against these groups. With the exception of two groups that work on human rights in multiple countries, the remaining eight groups focus on China and Tibet-related human rights issues. We focus on targeted malware typically delivered via e-mail that is specifically tailored to these groups as opposed to conventional spam which has been well characterized in numerous previous works [27, 42, 45, 52, 70, 71]. We consider the threats to these groups along two axes: the technical sophistication of the malware as well as sophistication of the social engineering used to deliver the malicious payload. We combine these two metrics to form an overall threat ranking that we call the Targeted Threat Index (TTI). While other scoring systems exist for characterizing the level of severity and danger of a technical vulnerability [7, 17, 41, 50], no common system exists for ranking the sophistication of targeted e-mail attacks. TTI allows us to gain insights into the relative sophistication of social engineering and malware leveraged against CSOs.

A key to the success of our study is a unique methodology, combining qualitative and technical analysis of

e-mails and their attachments with fieldwork (e.g. site visits) and interviews with affected CSOs. This methodology, which we describe in more detail in Section 3, allows us to both accurately rate the level of targeting of e-mail messages by interfacing with CSOs participating in our study (Section 4.2), and understand the relative technical sophistication of different malware families used in the attacks (Section 4.3). By combining the strengths of our qualitative and quantitative analysis, we are able to accurately understand trends in terms of social engineering and technical sophistication of politically-motivated targeted malware threats faced by CSOs.

Our study makes the following observations, which have implications for security strategies that CSOs can employ to protect themselves from targeted malware:

Attachments are the primary vector for email based targeted malware. More than 80% of malware delivered to Tibet-related organizations in our study and submitted to us is contained in an e-mail attachment. Further, for 2 of the 3 Tibetan organizations in our study (with at least 40 submitted e-mails), simply not opening attachments would mitigate more than 95% of targeted malware threats that use email as a vector.

Targeted malware technical sophistication is low. Social engineering sophistication is high We find that the technical sophistication of targeted malware delivered to CSOs in our study is relatively low (*e.g.,* relative to commercial malware that has been found targeting CSOs and journalists [35,36,38] and conventional financially motivated malware), with much more effort given to socially engineering messages to mislead users. This finding highlights the potential for education efforts focused on changing user behaviours rather than high-cost technical security solutions to help protect CSOs.

CSOs face persistent and highly motivated actors. For numerous malware samples in our study we observe several versions of the software appearing over the course of our four year study. These multiple versions show evidence of technical improvements to complement existing social engineering techniques.

Since the start of our study we have participated in a series of workshops with the participating Tibetan organizations to translate these results into a training curriculum. Specifically, we have educated them about how to identify suspicious e-mail headers to identify spoofed senders and demonstrated tools that can be used to check e-mailed links for malware and drive-by-downloads.

The rest of the paper is structured as follows. Section 2 presents relevant background on targeted malware and attacks on CSOs. Our data collection methodology is described in Section 3. We describe our targeting and technical sophistication metrics as well as how we combine them to produce the Targeted Threat Index (TTI) in Section 4. Training and outreach implications of our work are discussed in Section 5. We present related work in Section 6 and conclude in Section 7.

2 Background

2.1 Targeted Malware Overview

Targeted malware are a category of attacks that are distinct from common spam, phishing, and financially motivated malware. Spam and mass phishing attacks are indiscriminate in the selection of targets and are directed to the largest number of users possible. Similarly, financially motivated malware such as banking trojans seek to compromise as many users as possible to maximize the potential profits that can be made. The social engineering tactics and themes used by these kinds of attacks are generic and the attack vectors are sent in high volumes. By contrast targeted malware attacks are designed for specific targets, sent in lower volumes, and are motivated by the objective of stealing specific sensitive data from a target.

Targeted malware attacks typically involve the following stages [24, 66]:

Reconnaissance: During this stage attackers conduct research on targets including profiling systems, software, and information security defenses used to identify possible vulnerabilities and contextual information on personnel and activities to aid social engineering.

Delivery: During this stage a vector for delivering the attack is selected. Common vectors include e-mails with malicious documents or links, or contacting targets through instant messaging services and using social engineering to send malware to them. Typically, a target of such an attack receives an e-mail, possibly appearing to be from someone they know, containing text that urges the user to open an attached document (or visit a website).

Compromise: During this stage malicious code is executed on a target machine typically after a user initiated action such as opening a malicious document or link.

Command and Control: During this stage the infected host system establishes a communications channel to a command and control (C&C) server operated by the attackers. Once this channel has been established the attackers can issue commands and download further malware on to the system

Additional attacker actions: After a successful compromise is established, attackers can conduct a number of actions including ex-filtrating data from the infected host and transmitting it back to attackers through a process of encrypting, compressing, and transferring to a server

operated by the attackers. Attackers may also use peripherals such as webcams and microphones to monitor users in real time. The infected host may also serve as a starting point to infect other machines on the network and seek out specific information or credentials.

2.2 Targeted Malware and CSOs

Targeted malware has become recognized by governments and businesses around the world as a serious political and corporate espionage threat. The United States government has been particularly vocal on the threat targeted malware enabled espionage poses. General Keith Alexander, current Director of the National Security Agency and Commander of United States Cyber Command has stated that the theft of US intellectual property through cyber espionage constitutes the "greatest transfer of wealth in history" [47]. Recent widely publicized targeted malware intrusions against Google, RSA, the New York Times and other high profile targets have raised public awareness around these attacks [20, 44, 48]

Despite this increased attention, targeted malware is not a new problem, with over a decade of public reports on these kinds of attacks [66]. However, the majority of research on targeted malware is conducted by private security companies who typically focus on campaigns against industry and government entities. As a result, targeted attacks on civil society and non-governmental organizations have gone underreported despite the fact that these organizations have been shown to be frequently targeted by cyber espionage campaigns. In particular, communities related to ethnic minority groups in China including Tibetans, Uyghurs, and religious groups such as Falun Gong have been frequent targets of cyber espionage campaigns with reports dating back to at least 2002 [61].

In some cases, the same actors have been revealed to be targeting civil society groups, government and industry entities. A notable example of this was the 2009 report by the Citizen Lab, a research group at the University of Toronto, which uncovered the "GhostNet" cyber espionage network. GhostNet successfully compromised prominent organizations in the Tibetan community in addition to 1,295 hosts in 103 countries, including ministries of foreign affairs, embassies, international organizations, and news media [25]. The GhostNet case is not an isolated example, as other reports have shown CSOs (commonly Tibetan organizations) included as targets in campaigns that are also directed to a range of government and industry entities [8, 26, 28, 29, 54–56] Some of these reports include technical details on the CSO specific attacks [26, 28, 54, 55] while others note them as a target but do not address in detail [8, 29, 56].

While the majority of documented targeted malware campaigns against CSOs involve China and Tibet-related groups and potentially China-related attack operators [9–11, 23, 25, 26, 32, 61–65, 67, 68] , these kinds of attacks go beyond China. Recent research and news media have reported attacks against large human rights groups focused on multiple issues and countries [31, 46], and communities related to Syria [18] and Iran [37]. Researchers have also uncovered the use of commercial network intrusion products used to target activists from Bahrain [38], the United Arab Emirates [36], and journalists from Ethiopia [35].

3 Data collection

Since our study involves dealing with e-mail messages which may contain personally identifiable information (PII) and collection of information from CSOs who need to maintain privacy of their data, we consulted with our institutional research ethics board during the design of our study. The methods described below have been submitted to and approved by this board.

3.1 Study Participants

We recruited participants via three main channels: (1) an open call on our Web site, (2) outreach to organizations we had prior relationship with and (3) referrals from participating groups. As part of the study these groups agreed to share technical data (e.g., e-mails with suspicious attachments) and participate in interviews at the onset and end of the study. Their identity and any PII shared with us were kept strictly confidential.

For the purposes of our study, we focused on organizations with missions concerning the promotion or protection of human rights. For purposes of this study, "human rights" means any or all of the rights enumerated under the *Universal Declaration of Human Rights* [60], *the International Covenant on Civil and Political Rights* [58], and the *International Covenant on Economic, Social and Cultural Rights* [59]. We also considered organizations on a case by case basis that have a mission that does not directly implicate human rights, but who may nonetheless be targeted by politically motivated digital attacks because of work related to human rights issues (e.g., media organizations that report on human rights violations).

In total, 10 organizations participated in the study (summarized in Table 1). The majority of these groups work on China-related rights issues and five of these organizations focus specifically on Tibetan rights. The high rate of participation from China and Tibet-related human rights issues is due in part to our previous relationships with these communities and a significant interest and enthusiasm expressed by the groups. In addition to the China and Tibet-related groups, our study also includes

two groups, Rights Group 1 and 2 that work on multiple human rights related issues in various countries.

The majority of organizations operate from small offices with less than 20 employees. Some organizations (China Group 2, Tibet Group 2) have no physical office and consist of small virtual teams collaborating remotely, often from home offices. Of these groups only two (China Group 1, China Group 3) have a dedicated system administrator on staff. Other groups (Tibet Groups 1-5; China Group 2) rely on volunteers or staff with related technical skills (e.g. Web development) to provide technical support. Rights Group 1 and Rights Group 2 are much larger organizations relative to the others in our sample. Both organizations have over 100 employees, multiple offices, dedicated IT teams, and enterprise level computing infrastructures.

3.2 Data Sources

We collect the following pieces of information from the participant groups in order to understand the malware threats they face:

User-submitted e-mail messages. Our primary data source is a collection of e-mails identified by participants as suspicious which were forwarded to a dedicated e-mail server administered by our research team. When available these submissions included full headers, file attachments and / or links. There are three key limitations to relying on user-submitted e-mails for our analysis. First, we are only able to study e-mails identified by participants as suspicious, which may bias our results to only reporting threats that have been flagged by users. Further, individuals may forget to forward e-mails in some cases. Relying on self-reporting also creates bias between groups as individuals at different organizations may have different thresholds for reporting, which creates difficulties in accurately comparing submission rates between groups. Thus the amount of threat behaviour we see should be considered a lower bound on what occurs in practice. Second, having participants forward us e-mails does not allow us to verify if the targeted organization was successfully compromised by the attack (*e.g.,* if another member of the organization open and executed malware on their machine) and what the scope of the attack was. Finally, e-mail is only one vector that may be used to target organizations. Other vectors include waterhole attacks [21], denial of service attacks, or any other vectors (*e.g.,* physical threats like infected USB sticks). These limitations mean that it is possible that we did not comprehensively observe all attacks experienced by our study groups and some more advanced attacks may have gone unreported.

Recognizing the limitations of e-mail submissions, we complement user submitted emails with data from Network Intrusion Detection System (NIDS) alerts, website monitoring, and interviews. Also, upon request of study groups who were concerned of possible infection we analyzed packet capture data from suspect machines. Through the course of this supplementary analysis we did not find indications of malware compromise that used samples that were not included in our pool of user-submitted emails. In this paper we focus on reporting results from analyzing the user submitted emails through the TTI. The NIDS and website monitoring components were added later in our study and do not significantly contribute to TTI analysis. [1]

Table 2: Breakdown of e-mails submitted per group.

Organization Code	# of e-mails
China Group 1	53
China Group 2	18
China Group 3	58
Rights Group 1	28
Rights Group 2	2
Tibet Group 1	365
Tibet Group 2	177
Tibet Group 3	2
Tibet Group 4	97
Tibet Group 5	4

3.3 Overview of User-Submitted E-mails

The e-mails examined in this study span over four years, from October 14, 2009 to December 31, 2013. Data collection began on November 28, 2011, but China Group 3 and Tibet Group 1 forwarded us their pre-existing archives of suspicious emails, resulting in e-mail samples dating back to October 14, 2009. In total, we received 817 e-mails from the 10 groups participating in our study. Table 2 breaks down the submissions from each groups and illustrates that submissions were highly non-uniform across the groups. Thus, in general, we focus on the groups with at least 50 e-mail submissions for our analysis.

Figure 1 shows the cumulative number of e-mail submissions per month over the course of the study. For example, China Group 3 shared a set of e-mails received in 2010 by a highly targeted member of the organization, which can be observed in Figure 1. Tibet Group 1 accounts for the highest number of submissions relative to the other groups due to being one of the first groups in the study and being persistently targeted by politically motivated malware. Tibetan Groups 2 and 4, who joined the study later (in April 2012) show a similar submission rate to original Tibetan Group 1, suggesting these groups are targeted at a similar rate. In Section 4.2, we investi-

Table 1: Summary of groups participating in our study.

Organization Code	Description	Organization size
China Group 1	Human rights organization focused on rights and social justice issues related to China	Small (1-20 employees)
China Group 2	Independent news organization reporting on China	Small (1-20 employees)
China Group 3	Human rights organization focused on rights and social justice issues related to China	Small (1-20 employees)
Rights Group 1	Human rights organization focused on multiple issues and countries	Large (over 100 employees)
Rights Group 2	Human rights organization focused on multiple issues and countries	Large (over 100 employees)
Tibet Group 1	Human rights organization focused on Tibet	Small (1-20 employees)
Tibet Group 2	Human rights organization focused on Tibet	Small (1-20 employees)
Tibet Group 3	Independent news organization reporting on Tibet	Small (1-20 employees)
Tibet Group 4	Human rights organization focused on Tibet	Small (1-20 employees)
Tibet Group 5	Human rights organization focused on Tibet	Small (1-20 employees)

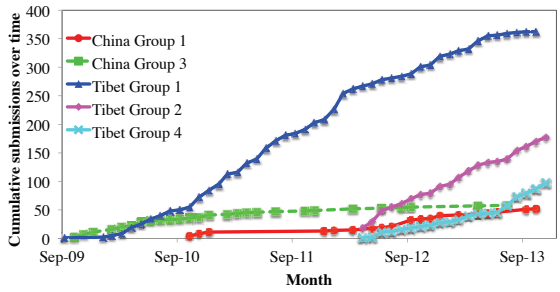


Figure 1: Cumulative number of messages per group over the course of our study for groups that submitted at least 50 e-mail messages.

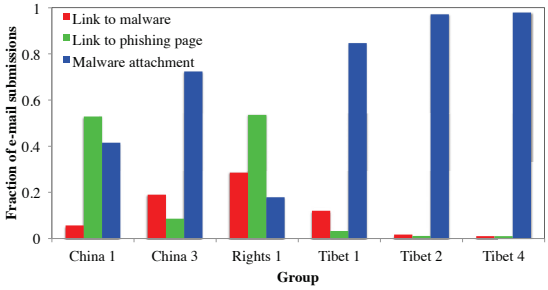


Figure 2: Breakdown of malicious e-mails based on whether they deliver malware as an attachment, refer the use to a link with a malicious file, or attempt to phish data from the user.

gate commonalities in targeting of these groups.

We further classify e-mails as malicious if they include attached malware, a direct link to malware or a site with a drive-by download, or a link to a phishing page. Figure 2 shows the amount of e-mails of each type for the groups that submitted at least 25 e-mails to our system. The most common approach employed in these e-mails was attaching a malicious payload to the e-mail itself. However, we notice a higher rate of phishing attacks on the China-related groups and the rights groups working on multiple international human rights issues. In particular, 46% of the e-mails submitted by China Group 1, and 50% of the e-mails submitted by Rights Group 1, direct the user to a phishing Web site. In the case of China Group 1, this large proportion of phishing sites is observed because this group configured their spam filter to forward e-mails to our system, resulting in us receiving a large number of generic, non-targeted spam. In contrast, the phishing observed for Rights Group 1, while low in volume (13 out of 26 messages) is targeted. We delve more into how we rate the targeting of e-mails in Section 4.2.

The rate of submissions to our project meant that it was feasible to manually analyze e-mail attachments for malware as they were submitted. This analysis gives us higher confidence in our results because AV signatures are frequently unable to detect new or modified threats, and can overlook the presence of a malicious payload that can be easily identified upon manual inspection (e.g. shellcode in an RTF exploit). In total, we analyzed 3,617 payload files and found 2,814 (78%) of them to be malicious. Section 4.3 describes our analysis methodology in more detail.

4 Targeted Threat Index

Our dataset includes a wide range of targeted malware threats varying in level of both social engineering and technical complexity. This range presents a challenge in ranking the relative sophistication of the malware and targeting tactics used by attackers.

While scoring systems such as the Common Vulnerability Scoring System [17] exist for the purpose of communicating the level of severity and danger of a vulnerability, there is no standardized system for ranking

the sophistication of targeted email attacks. This gap is likely because evaluating the sophistication of the targeting is non-technical, and cannot be automated due to the requirement of a strong familiarity with the underlying subject material.

To address this gap we developed the Targeted Threat Index (TTI) to assign a ranking score to the targeted malicious emails in our dataset. The TTI score is intended for use in prioritizing the analysis of incoming threats, as well as for getting an overall idea of how severely an organization is threatened.

The TTI score is calculated by taking a base value determined by the sophistication of the targeting method, which is then multiplied by a value for the technical sophistication of the malware. The base score can be used independently to compare emails, and the combined score gives an indication of the level of effort an attacker has put into individual threats.

4.1 TTI Metric

The TTI score is calculated in two parts:

(*Social Engineering Sophistication Base Value*)
$\times$(*Technical Sophistication Multiplier*) = *TTI Score*

TTI scores range from 1 to 10, where 10 is the most sophisticated attack. Scores of 0 are reserved for threats that are not targeted, even if they are malicious. For example, spam using an attached PDF or XLS to bypass anti-spam filters, and highly sophisticated financially motivated malware, would both score 0.

This section overviews how we compute the *Social Engineering Sophistication Base Value* (Section 4.2) and the *Technical Sophistication Multiplier* (Section 4.3). In Section 4.4, we present the results of computing and analyzing the TTI value of threats observed by the organizations in our study. We also discuss implications and limitations of the metric.

4.2 Social Engineering Tactics

We leverage a manual coding approach to measure the sophistication of social engineering tactics used in the attacks observed by the organizations in our study. While automated approaches may be explored in the future, this manual analysis allows us to have high confidence in our results, especially since understanding the social engineering often required contextual information provided by the organizations in our study. To quantify the level of sophistication, we manually analyse the e-mail subject line, body, attachments and header fields. We perform an initial content analysis by coding the e-mails based on their semantic content, and then use these results to generate a numerical metric quantifying the level of targeting used.

4.2.1 Content coding and analysis results

We code the e-mails based on their subject line, body, attachments and headers using the following methodology:

Subject line, body, and attachments. The content of the subject line, body and attachments for each submitted e-mail were content coded into 8 themes, each containing categories for specific instances of the theme: Country / Region (referring to a specific geographical country or region); Ethnic Groups (referring to a specific ethnic group); Event (referring to a specific event); Organizations (referring to specific organizations); People (referring to specific persons), Political (reference to specific political issues), Technology (reference to technical support), Miscellaneous (content without clear context or categories that do not fall into one of the other themes). Table 3 summarizes the themes and provides examples of categories within each theme.

E-mail headers. The header of each e-mail was analyzed to determine if the sending e-mail address was spoofed or the e-mail address was otherwise designed to appear to come from a real person and / or organization (e.g. by registering an e-mail account that resembles a person and / or organization's name from a free mail provider). We divide the results based on whether they attempted to spoof an organization or a specific person.

Using this manual analysis, we perform a content analysis of e-mails submitted by the organizations. Results of this analysis confirm that social engineering is an important tool in the arsenal of adversaries who aim to deliver targeted malware. Specifically, 95% and 97% of e-mails to Chinese and Tibetan groups, respectively, included reference to relevant regional issues. Spoofing of specific senders and organizations was also prevalent with 52% of e-mails to Tibetan groups designed to appear to come from real organizations, often from within the Tibetan community. For example, a common target of spoofing was the Central Tibetan Administration (CTA), referenced in 21% of the spoofed e-mails, which administers programs for Tibetan refugees living in India and advocates for human rights in Tibet. While the number of e-mail submissions were lower for the general human rights groups, we observe similar trends there with 92% of e-mails submitted by Rights Group 1 appearing to come from individuals in the group (as a result of spoofing).

In some cases we even observed the same attackers targeting multiple CSOs with customized e-mail lures. For example, we tracked a campaign that targeted China Groups 1 and 2, and Tibet Group 1 with a remote access

Table 3: Overview of themes and categories within the themes for grouping targeted e-mail messages.

Theme	Total Categories	Example Categories
Country/Region	26	China, US, European Union
Ethnic Groups	2	Tibetan, Uyghur
Event	31	self immolation, Communist Party of China, 18th National Party Congress
Organizations	32	United Nations, Central Tibetan Administration
People	31	His Holiness the Dalai Lama, Hu Jintao
Political	6	human rights, terrorism
Technology	5	software updates, virtual private servers
Miscellaneous	1	content without clear context which falls outside of the other themes

trojan we call IEXPL0RE [22] China Group 1 received the malware in e-mails claiming to be from personal friends whereas China Group 2 received the malware in an e-mail containing a story about a high-rise apartment building fire in China. In contrast, Tibet Group 1 received the malware embedded into a video of a speech by the Dalai Lama, attached to an e-mail about a year in review of Tibetan human rights issues.

4.2.2 Social Engineering Sophistication Base Value

While the content analysis results clearly show attacks tailored to the interests of targeted groups, content coding alone does not give a relative score of the sophistication used in the attacks. We now describe how we assign the "social engineering sophistication base value" to e-mails based on their level of social engineering.

To measure the targeting sophistication we assign a score that ranges from 0-5 that rates the social engineering techniques used to get the victim to open the attachment. This score considers the content and presentation of the e-mail message as well as the claimed sender identity. This determination also includes the content of any associated files, as malware is often implanted into legitimate relevant documents to evade suspicion from users when the malicious documents are opened.

The Social Engineering Sophistication Base Value is assigned based on the following criteria:

0 Not Targeted: Recipient does not appear to be a specific target. Content is not relevant to the recipient. The e-mail is likely spam or a non-targeted phishing attempt.

1 Targeted Not Customized: Recipient is a specific target. Content is not relevant to the recipient or contains information that is obviously false with little to no validation required by the recipient. The e-mail header and/or signature do not reference a real person or organization.

2 Targeted Poorly Customized: Recipient is a specific target. Content is generally relevant to the target but has attributes that make it appear questionable (e.g. incomplete text, poor spelling and grammar, incorrect addressing). The e-mail header and / or signature may reference a real person or organization.

3 Targeted Customized: Recipient is a specific target. Content is relevant to the target and may repurpose legitimate information (such as a news article, press release, conference or event website) and can be externally verified (e.g. message references information that can be found on a website). Or, the e-mail text appears to repurpose legitimate e-mail messages that may have been collected from public mailing lists or from compromised accounts. The e-mail header and / or signature references a real person or organization.

4 Targeted Personalized: Recipient is a specific target. The e-mail message is personalized for the recipient or target organization (e.g. specifically addressed or referring to individual and / or organization by name). Content is relevant to the target and may repurpose legitimate information that can be externally verified or appears to repurpose legitimate messages. The e-mail header and / or signature references a real person or organization.

5 Targeted Highly Personalized: Recipient is a specific target. The e-mail message is individually personalized and customized for the recipient and references confidential / sensitive information that is directly relevant to the target (e.g. internal meeting minutes, compromised communications from the organization). The e-mail header and / or signature references a real person or organization.

Content coding of emails and determinations of social engineering ratings for the TTI were performed by five independent coders who were given a code book for content categories and the TTI social engineering scale with examples to guide analysis. We performed regular inter-rater reliability checks and flagged any potential edge cases and inconsistencies for discussion and re-evaluation. Following completion of this analysis, two of the authors reviewed the social engineering base value scores to ensure consistency and conformity to the scale. We provide specific examples of each of these targeting values in Appendix A.

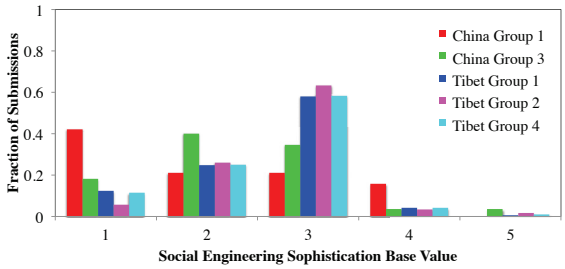


Figure 3: Social engineering sophistication base value assigned to e-mail submissions from groups that submitted at least 50 e-mails.

4.2.3 Summary of Social Engineering Sophistication Base Value

Figure 3 shows the targeting score for organizations in our study who submitted at least 50 e-mails. We can see that actors targeting these groups put significant effort into targeting their messages, in particular the three Tibetan groups included in Figure 3 observe more than half of their messages with a targeting score of 3 or higher. This result means adversaries are taking care to make the e-mail appear to come from a legitimate individual or organization, and include relevant information (*e.g.,* news reports or exchanges from public mailing lists). Higher targeting scores, which result from actions such as personalizing lures to an individual in the group, or including information that requires prior reconnaissance tend to be more rare, but we do observe instances of them. For example, in the case of China Group 3, we observed an e-mail which received a social engineering score of 5, which claimed to be from the group's funder and referenced a specific meeting they had planned that was not public knowledge.

4.3 Technical Sophistication

We manually analyzed all submitted emails and attachments to determine whether they contained politically-motivated malware. The malware is then analyzed in detail to extract information such as the vulnerability, C&C server (if present), and technical sophistication of the exploit.

4.3.1 Assessment methodology

The first step in our analysis pipeline is determining whether the email contains politically motivated malware or not. This process involves an initial inspection for social engineering of the email message and attachment (e.g., an executable pretending to be a document). We also correlate with other emails received as part of this project to identify already-known malware. Well-known malware attacks (*e.g.,* the Zeus trojan masquerading as an email from the ACH credit card payment processor, or Bredolab malware pretending to be from the DHL courier service) are not considered targeted attacks in our study, but are still kept for potential review.

Once we have identified emails which we suspect of containing politically-motivated malware, we perform the following analysis steps on any attachments to verify that they indeed contain malware. First, we run the attachment in a sandboxed VM to look for malicious activity *e.g.,* an Office document writing files to disk or trying to connect to a C&C server. We also check the MD5 hash of the attachment against the Virus Total database to see if it matches existing viruses. We also manually examine the attached file for signs of malicious intent (*e.g.,* executable payload in a PDF, shellcode or Javascript). We exclude any graphics attached to the email which are used for social engineering (and do not contain malicious payload) from our analysis.

We follow this initial analysis with more detailed technical analysis of the attachments which we confirm contain malware. First, we manually verify the file type of the attachment for overview statistics. This manual analysis is necessary as the Unix file command may be misled by methods of manipulating important bytes in the file (*e.g.,* replacing \rtf1 with \rtf[null]). We then identify if the vulnerability included in the malware already exists in a corpus of vulnerabilities, such as the Common Vulnerabilities and Exposures (CVE) naming system. We also perform analysis of network traffic from the attachment to identify the C&C server the malware attempts to contact. In cases where the malware does not execute in our controlled environment we manually examine the file to extract the relevant information.

On a case-by-case basis we use additional tools such as IDA [1] and OllyDbg [3] for detailed static and dynamic analysis, respectively. Our goal in this analysis is to identify relationships between malware campaigns between organizations, or instances of the same malware family repeatedly targeting a given organization. By observing overlapping C&C servers, or mapping malware to common exploits identified by anti virus/security companies we can cluster attacks that we believe come from the same malware family and potentially the same adversary.

4.3.2 Technical Sophistication Multiplier

While the previous analysis is useful for understanding the nature of threats, we also score threats numerically to aid in understanding the relative technical sophistication of their approaches. Each malware sample is assigned one of the following values:

1 Not Protected - The sample contains no code protec-

tion such as packing, obfuscation (e.g. simple rotation of interesting or identifying strings), or anti-reversing tricks.

1.25 Minor Protection - The sample contains a simple method of protection, such as one of the following: code protection using publicly available tools where the reverse method is available, such as UPX packing; simple anti-reversing techniques such as not using import tables, or a call to IsDebuggerPresent(); self-disabling in the presence of AV software.

1.5 Multiple Minor Protection Techniques - The sample contains multiple distinct minor code protection techniques (anti-reversing tricks, packing, VM / reversing tools detection) that require some low-level knowledge. This level includes malware where code that contains the core functionality of the program is decrypted only in memory.

1.75 Advanced Protection - The sample contains minor code protection techniques along with at least one advanced protection method such as rootkit functionality or a custom virtualized packer.

2 Multiple Advanced Protection Techniques - The sample contains multiple distinct advanced protection techniques, e.g. rootkit capability, virtualized packer, multiple anti-reversing techniques, and is clearly designed by a professional software engineering team.

The purpose of the technical sophistication multiplier is to measure how well the payload of the malware can conceal its presence on a compromised machine. We use a multiplier because advanced malware requires significantly more time and effort (or money, in the case of commercial solutions) to customize for a particular target.

We focus on the level of obfuscation used to hide program functionality and avoid detection for the following reasons: (1) It allows the compromised system to remain infected for a longer period; (2) it hinders analysts from dissecting a sample and developing instructions to detect the malware and disinfect a compromised system; (3) since most common used remote access trojans (RATs) have the same core functionality (e.g. keylogging, running commands, exfiltrating data, controlling microphones and webcams, etc.) the level of obfuscation used to conceal what the malware is doing can be used to distinguish one RAT from another.

4.3.3 Summary of Technical Sophistication Multiplier Value

Figure 4 shows the technical sophistication multiplier values for e-mails submitted by the different organizations in our study. One key observation we make here is that the email-based targeted malware that was self-reported by our study groups is relatively simple. The highest multiplier value we see is 1.5 and even that value is seen infrequently. The majority of malware observed is rated either 1 or 1.25 according to our technical scoring criteria, with Tibetan Groups observing a higher fraction of malware rated 1.25 and Chinese groups observing a higher fraction rated 1.

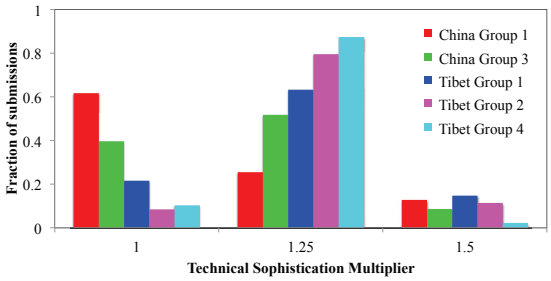


Figure 4: Technical sophistication multiplier assigned to e-mail submissions from groups that submitted at least 50 e-mails.

The technical sophistication multiplier value is also useful for assessing the technical evolution of threats in our study. When we group malware into different family groups we can see some of these groups are under active development. For example, we observe multiple versions of the Enfal [40, 49], Mongal [14], and Gh0st RAT [15] families with increasing levels of sophistication and defenses in place to protect the malware code (resulting in an increase in technical multiplier from 1 to 1.25 for these families). Since our technical multiplier value focuses on how well malware code defends and disguises itself, changes to other aspects of the code may not result in an increase in value (*e.g.,* we observe multiple versions of the IMuler.A/Revir.A malware which all receive a score of 1). Interestingly, when we observe both a Windows and Mac version of a given malware family, the technical score for the Mac version tended to be lower with the Mac version being relatively primitive relative to the Windows variant.

4.4 TTI Results

We now show how the TTI metric can help us better characterize the relative threat posed by targeted malware. Figure 5 shows the technical sophistication multiplier and maximum/minimum TTI scores for malware families observed in our dataset. Since we primarily observe simple malware, with a technical sophistication multiplier of 1 or 1.25, this value does a poor job of differentiating the threat posed by the different malware families to the CSOs. However, by incorporating both the technical sophistication and targeting base value into the TTI metric we can gain more insights into how effective these

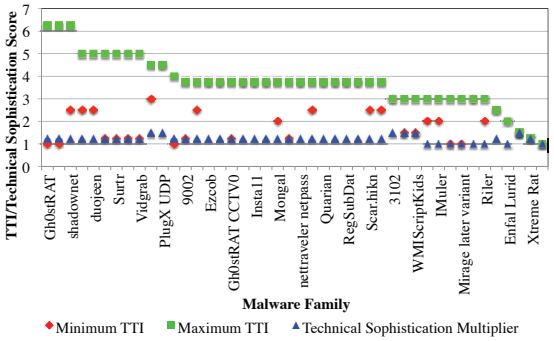


Figure 5: Comparison of the maximum and minimum TTI score and technical sophistication multiplied for malware families observed in our data (sorted in decreasing order of maximum TTI).

Table 4: Top malware families in our data set in terms of technical sophistication multiplier and in terms of final TTI score.

Technical Sophistication		
Family	TTI	Tech. Soph.
3102	3	1.5
nAspyUpdate	1.5	1.5
PlugX	4.5	1.5
PoisonIvy	3	1.5
WMIScriptKids	3	1.5
TTI		
Family	TTI	Tech. Soph. .
Gh0stRAT LURK0	6.25	1.25
shadownet	6.25	1.25
conime	5	1.25
duojeen	5	1.25
iexpl0re	5	1.25

threats may be in practice.

The impact of using TTI is especially apparent when trying to gain insights into the targeted malware that poses the biggest risk to CSOs. Table 4 shows the top 5 malware families we observe in terms of technical sophistication and in terms of TTI score. If we consider the malware families with the highest technical sophistication, we can see that their TTI values are relatively low, with maximums ranging from 1.5 to 4.5. These tend to be malware families that are familiar to researchers. In particular, PlugX and PoisonIvy have been used in targeted attacks together [43] and PlugX is still actively used and under constant development [16]. Despite technical sophistication, the social engineering lures of these threats are not well crafted and pose less of a risk to the CSOs whose members may be able to identify and avoid these threats.

In contrast, the top 5 malware families in terms of TTI have lower technical sophistication (1.25) but much higher levels of social engineering. It is no surprise that threats which score the highest TTI use well known malware that have been extensively documented in attacks against a variety of targets. For example, the TTI scores reflect that Gh0st RAT continues to be seen in higher risk attacks due to its popularity amongst attackers even though it is an older and not particularly advanced tool. Since there is no direct connection between the technical sophistication of threats and the level of social engineering used to target CSOs, it is likely that different threat actors, with a different focus, are at work here. Indeed, Gh0st RAT was discovered by the Citizen Lab in their analysis of GhostNet [25] and IEXPL0RE RAT was discovered and named for the first time in our work.

Another observation is that commercial malware such as FinFisher and DaVinci RCS, while being of much higher technical sophistication (relative to the samples in our study), do not necessarily score higher on TTI than a targeted attack with advanced social engineering and more basic malware. For example, analyzing a FinFisher sample targeted against Bahraini activists [38] with the TTI, produces an overall TTI score that is dependent on the social targeting aspect, even though the malware is very technically advanced. In this case, the FinFisher attack scores 4.0 on the TTI (base targeting score of 2 with a technical multiplier of 2). Although the email used in the attack references the name and organization of a real journalist, the content is poorly customized, and has attributes that look questionable. However, the technical sophistication of the malware is advanced earning it a score of 2 due to multiple advanced protection techniques, including a custom-written virtualized packer, MBR modification, and rootkit functionality. The sample also uses multiple minor forms of protection, including at least half a dozen anti-debugging tricks. Even though the technical multiplier is the maximum value, the overall TTI score is only 4.0 due to the low targeting base value. FinFisher is only effective if it is surreptitiously installed on a users' computer. If the malware is delivered through an email attachment, infection is only successful if the user opens the malicious file. The advanced nature of this malware will cause the overall score to increase quickly with improved targeting, but as it still requires user intervention, this threat scores lower overall than attacks with highly targeted social engineering using less sophisticated malware.

Similar findings can also be observed in attacks using DaVinci RCS developed by Italy-based company Hacking Team against activists and independent media groups from the United Arab Emirates and Morocco [36]. While the malware used in these publicly reported attacks is

technically sophisticated, the social engineering lures employed are poorly customized for the targets resulting in a 4.0 TTI score (targeting base value 2, technical multiplier 2).

These results support the idea that different threat actors have varying focuses and levels of resources, and as a result, different methodologies for attacks. For example, the majority of malware submitted by our study groups appear to be from adversaries that have in-house malware development capabilities and the capacity to organize and implement targeted malware campaigns. These adversaries are spending significant effort on social engineering, but generally do not use technically advanced malware. Conversely, the adversaries using FinFisher and DaVinci RCS have bought these products rather than develop malware themselves. However, while the FinFisher and RCS samples are technically sophisticated pieces of malware, the attacks we analyzed are not sophisticated in terms of social engineering tactics.

4.5 Limitations of TTI

While the Targeted Threat Index gives insight into the distribution of how sophisticated threats are, we are still in the process of evaluating and refining it through interactions with the groups in our study and inclusion of more sophisticated threats observed in related investigations in our lab. Average TTI scores in our dataset may be skewed due to the self-reporting method we use in the study. Very good threats are less likely to be noticed and reported while being sent to far fewer people, and low-quality emails are much more likely to be sent in bulk and stand out. It is also possible that individuals in different groups may be more diligent in submitting samples, which could affect between group comparisons. We are more interested, however, in worst-case (highest) scores and not in comparing the average threat severity between organizations.

Finally, this metric is calculated based on the technical sophistication of the payload, not on the specific exploit. There is currently no method to modify the TTI score in a way similar to the temporal metrics used by the CVSS metric. A temporal metric could be added to increase the final TTI value for 0-day vulnerabilities, or possibly to reduce the score for exploits that are easily detectable due to a public and well-known generation script, e.g. Metasploit [2].

5 Implications

Our study primarily focuses on threats that groups working on human rights issues related to Tibet or China are currently facing. While our dataset is concentrated on these types of groups, our results have implications for how CSOs can protect themselves against email-based targeted malware.

Specifically, we find that moving towards cloud-based platforms (*e.g.,* Google Docs) instead of relying on e-mail attachments would prevent more than 95% of the e-mail malware seen by 2 out of 3 Tibetan groups that had more than 50 e-mail submissions.

Further, our results highlight the potential for lower-cost user education initiatives to guard against sophisticated social engineering attacks, rather than high cost technical solutions. This observation stems from the fact that much of the malware we observe is not technically sophisticated, but rather relies on social engineering to deliver its payload by convincing users to open malicious attachments or links. Other studies [35, 36, 38] that have revealed the use of commercial malware products against CSOs and journalists have shown that many of these cases also rely on duping users into opening malicious e-mail attachments or social engineered instant messaging conversations. These incidents show that even advanced targeted malware requires successful exploitation of users through social engineering tactics.

User education can be a powerful tool against the kinds of targeted attacks we observed in this study. Indeed, the Tibetan community has taken an active approach with campaigns that urge Tibetan users to not send or open attachments and suggests alternative cloud based options such as Google Docs and Dropbox for sharing documents [53]. We have also engaged the Tibetan groups in a series of workshops to introduce training curriculum which draws on examples submitted by organizations participating in our study. We have also provided them with technical background to identify suspicious e-mail headers and how to use free services to check the validity of suspicious links in e-mail messages.

The mitigation strategies presented here are focused on email vectors and do not consider all of the possible attacks these groups may face. We highlight these strategies in particular because the majority of groups in our study identified document-based targeted malware as a high priority information security concern. The adversaries behind these attacks are highly motivated and will likely adapt their tactics as users change their behaviors. For example, it is plausible that if every user in a particular community began to avoid opening attachments and document-based malware infected fewer targets, attackers may move on to vectors such as waterhole attacks or attacks on cloud document platforms to fill the gap. User education and awareness raising activities need to be ongoing efforts that are informed by current research on the state of threats particular communities are experiencing. Evaluation of the effectiveness of user education efforts in at risk communities and corresponding reactions from attackers is required to understand the dynamics between

these processes.

6 Related Work

There is a wide body of literature on filtering and detection methods for spam [27,42,45,52,70,71] and phishing emails and websites [12, 34, 39, 69]. Attention has also been given to evaluating user behavior around phishing attacks and techniques for evading them [6, 30, 33]. By comparison research on detecting email vectors used for targeted malware attacks is limited. A notable exception is [4, 5], which uses threat and recipient features with a random forest classifier to detect targeted malicious emails in a dataset from a large Fortune 500 company. Other work has focused on imporoving detection of documents (e.g. PDF, Microsoft Office) with embedded malicious code [13, 51, 57]

Another area of research explores methods for modeling the stages of targeted attacks and using these models to develop defenses. Guira and Wang [19] propose a conceptual attack model called the attack pyramid to model targeted attacks and identify features that can be detected at the various stages. Hutchins, Cloppert and Amin, [24] use a kill chain model to track targeted attack campaigns and inform defensive strategies.

Metrics have been developed to characterize security vulnerabilities and their severity [7, 41, 50]. The industry standard is the Common Vulnerability Scoring System (CVSS) [17], which uses three metric groups for characterizing vulnerabilities and their impacts. These groups are: base metric group (the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments), temporal metric group (characteristics of a vulnerability that change over time but not among user environments) and environmental metric group (characteristics of a vulnerability that are relevant and unique to a particular user's environment). The CVSS is a widely adopted metric, but only rates technical vulnerabilities. Targeted attacks rely on a user action of opening a malicious attachment or visiting a malicious link to successfully compromise a system. Therefore, the sophistication of message lures and other social engineering tactics are an important part of determining the severity of a targeted attack. Systems like the CVSS cannot address this contextual component.

Our study makes the following contributions to the literature. Previous studies of targeted attacks against CSOs usually focus on particular incidents or campaigns and do not include longitudinal observations of attacks against a range of CSO targets. While standards exist for rating the sophistication of technical vulnerabilities and research has been done on detecting targeted malware attacks and modeling campaigns, there is no scoring system that considers both the sophistication of malware and social engineering tactics used in targeted malware attacks. We address this gap through development of the TTI and validate the metric against four years of data collected from 10 CSOs.

7 Conclusions

Our study provides an in-depth look at targeted malware threats faced by CSOs. We find that considering the technical sophistication of these threats alone is insufficient and that educating users about social engineering tactics used by adversaries can be a powerful tool for improving the security of these organizations. Our results point to simple steps groups can take to protect themselves from document-based targeted malware such as shifting to cloud-based document platforms instead of relying on attachments which can contain exploits. Further research is needed to measure the effectiveness of education strategies for changing user behaviour and how effective these efforts are in mitigation of document-based malware for CSOs. Further work is also required in monitoring how attackers adapt tactics in response to observed behavioural changes in targeted communities.

In ongoing work we are continuing our collection of e-mails and NIDS alerts as well as monitoring other attacks against these groups (*e.g.,* waterhole attacks and DoS attacks) to understand how threats vary based on their delivery mechanism. We are also working to extend our methodology to more diverse CSO communities such as those in Latin America, Africa, and other underreported regions to better document the politically motivated digital threats they may be experiencing.

Acknowlegements

This work was supported by the John D. and Catherine T. MacArthur Foundation. We are grateful to Jakub Dalek, Sarah McKune, and Justin Wong for research assistance. We thank the USENIX Security reviewers and our shepherd Prof. J. Alex Halderman for helpful comments and guidance. We are especially grateful to the groups who participated in our study.

References

[1] https://www.hex-rays.com/products/ida/.

[2] http://www.metasploit.com/.

[3] http://www.ollydbg.de/.

[4] AMIN, R. M. *Detecting Targeted Malicious Emails Through Supervised Classification of Persistent Threat and Recipient Oriented Features*. Doctor of philisophy, George Washington University, 2011.

[5] AMIN, R. M., RYAN, J., H, J. C., AND VAN DORP, J. R. Detecting Targeted Malicious Email. *IEEE Security & Privacy 10*, 3 (2012), 64–71.

[6] BLYTHE, M., PETRIE, H., AND CLARK, J. A. F for Fake: Four Studies on How We Fall for Phish. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 3469–3478.

[7] CERT. Vulnerability Notes Database Field Descriptions, 2014.

[8] CHIEN, E., AND O'GORMAN, G. The Nitro Attack: Stealing Secrets from the Chemical Industry. Tech. rep., Symantec, 2011.

[9] CITIZEN LAB. Information Operations and Tibetan Rights in the Wake of Self-Immolations: Part I. Tech. rep., University of Toronto, 2012.

[10] CITIZEN LAB. Recent Observations in Tibet-Related Information Operations: Advanced social engineering for the distribution of LURK malware. Tech. rep., University of Toronto, 2012.

[11] CITIZEN LAB. Permission to Spy: An Analysis of Android Malware Targeting Tibetans. Tech. rep., University of Toronto, 2013.

[12] COVA, M., KRUEGEL, C., AND VIGNA, G. There is no free phish: an analysis of free and live phishing kits. In *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies* (July 2008), USENIX Association, p. 4.

[13] CROSS, J. S., AND MUNSON, M. A. Deep pdf parsing to extract features for detecting embedded malware. Tech. rep., Sandia National Laboratories, 2011.

[14] DEEP END RESEARCH. Library of Malware Traffic Patterns, 2013.

[15] FAGERLAND, S. The Many Faces of Gh0st Rat. Tech. rep., Norman, 2012.

[16] FAGERLAND, S. PlugX used against Mongolian targets. Tech. rep., 2013.

[17] FIRST. Common Vulnerability Scoring System (CVSS-SIG), 2007.

[18] GALPERIN, EVA, MARQUIS-BOIRE, MORGAN, SCOTT-RAILTON, J. Quantum of Surveillance: Familiar Actors and Possible False Flags in Syrian Malware Campaigns — Electronic Frontier Foundation. Tech. rep., Electronic Frontier Foundation and The Citizen Lab, University of Toronto.

[19] GIURA, P., AND WANG, W. A Context-Based Detection Framework for Advanced Persistent Threats. *International Conference on Cyber Security (CyberSecurity) 0* (2012), 69–74.

[20] GOOGLE. A new approach to China, 2012.

[21] GRAGIDO, W. Lions at the Watering Hole: The VOHO Affair. Tech. rep., RSA, 2012.

[22] HARDY, S. IEXPL0RE RAT. Tech. rep., Citizen Lab, University of Toronto, 2012.

[23] HARDY, SETH KLEEMOLA, K. Surtr: Malware Family Targeting the Tibetan Community. Tech. rep., Citizen Lab, University of Toronto, 2013.

[24] HUTCHINS, E. M., CLOPPERT, M. J., AND AMIN, R. M. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. In *6th International Conference on Information Warfare and Security* (2011).

[25] INFORMATION WARFARE MONITOR. Tracking GhostNet: Investigating a Cyber Espionage Network. Tech. rep., University of Toronto, 2009.

[26] INFORMATION WARFARE MONITOR. Shadows in the Cloud: Investigating Cyber Espionage 2.0. Tech. rep., University of Toronto, 2010.

[27] JUNG, J., AND SIT, E. An Empirical Study of Spam Traffic and the Use of DNS Black Lists. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2004), IMC '04, ACM, pp. 370–375.

[28] KASPERSKY LAB. The NetTraveler Attacks. Tech. rep., Trend Micro, 2013.

[29] KASPERSKY LAB. Unveiling "Careto" - The Masked APT. Tech. rep., 2014.

[30] KIRLAPPOS, I., AND SASSE, M.-A. Security Education against Phishing: A Modest Proposal for a Major Rethink. *Security Privacy, IEEE 10*, 2 (Mar. 2012), 24–32.

[31] KREBS, B. Espionage Hackers Target Watering Hole Sites, 2012.

[32] LI, F., LAI, A., AND DDL, D. Evidence of Advanced Persistent Threat: A case study of malware for political espionage. In *2011 6th International Conference on Malicious and Unwanted Software* (Oct. 2011), IEEE, pp. 102–109.

[33] LIN, E., GREENBERG, S., TROTTER, E., MA, D., AND AYCOCK, J. Does Domain Highlighting Help People Identify Phishing Sites? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 2075–2084.

[34] MAIORCA, D., CORONA, I., AND GIACINTO, G. Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 119–130.

[35] MARCZAK, B., GUARNIERI, C., MARQUIS-BOIRE, M., AND SCOTT-RAILTON, J. Hacking Team and the Targeting of Ethiopian Journalists. Tech. rep., Citizen Lab, University of Toronto, 2014.

[36] MARQUIS-BOIRE, M. Backdoors are Forever: Hacking Team and the Targeting of Dissent. Tech. rep., Citizen Lab, University of Toronto, 2013.

[37] MARQUIS-BOIRE, M. Iranian anti-censorship software 'Simurgh' circulated with malicious backdoor. Tech. rep., Citizen Lab, University of Toronto, 2013.

[38] MARQUIS-BOIRE, M., MARCZAK, B., GUARNIERI, C., AND SCOTT-RAILTON, J. For Their Eyes Only: The Commercialization of Digital Spying. Tech. rep., Citizen Lab, University of Toronto, 2013.

[39] MAURER, M.-E., AND HÖFER, L. Sophisticated Phishers Make More Spelling Mistakes: Using URL Similarity against Phishing. In *CSS* (2012), pp. 414–426.

[40] MCAFEE. Enfal, 2008.

[41] MICROSOFT CORPORATION. Security Bulletin Severity Rating System, 2012.

[42] PANTEL, P., AND LIN, D. SpamCop: A Spam Classification & Organization Program. In *Learning for Text Categorization: Papers from the 1998 Workshop* (1998), pp. 95–98.

[43] PAZ, R. D. PlugX: New Tool For a Not So New Campaign. Tech. rep., Trend Micro, 2012.

[44] PERLROTH, N. Chinese Hackers Infiltrate New York Times Computers, Jan. 2013.

[45] RAMACHANDRAN, A., FEAMSTER, N., AND VEMPALA, S. Filtering spam with behavioral blacklisting. In *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07* (New York, New York, USA, Oct. 2007), ACM Press, p. 342.

[46] RILEY, M., AND LAWRENCE, D. Hackers Linked to Chinas Army Seen From EU to D.C., 2012.

[47] ROGIN, J. NSA Chief: Cybercrime constitutes the greatest transfer of wealth in history. *Foreign Policy* (2012).

[48] RSA. Anatomy of an Attack.

[49] SANCHO, DAVID, VILLENEUVE, N. LURID: Attribution Isn't Easy. Tech. rep., Trend Micro.

[50] SANS. @Risk: The Consensus Security Alert, 2014.

[51] SMUTZ, C., AND STAVROU, A. Malicious PDF Detection Using Metadata and Structural Features. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 239–248.

[52] TAYLOR, B. Sender Reputation in a Large Webmail Service. In *Third Conference on Email and Anti-Spam (CEAS 2006)* (2006).

[53] TIBET ACTION INSTITUTE. https://tibetaction.net/detach-from-attachments/.

[54] TREND MICRO. IXESHE: An APT campaign. Tech. rep., 2012.

[55] TREND MICRO. Luckycat Redux: Inside an APT campaign with multiple targets in India and Japan. Tech. rep., 2012.

[56] TREND MICRO. 2Q Report on Targeted Attack Campaigns. Tech. rep., 2013.

[57] TZERMIAS, Z., SYKIOTAKIS, G., POLYCHRONAKIS, M., AND MARKATOS, E. P. Combining Static and Dynamic Analysis for the Detection of Malicious Documents. In *Proceedings of the Fourth European Workshop on System Security* (New York, NY, USA, 2011), EUROSEC '11, ACM, pp. 4:1—-4:6.

[58] UNITED NATIONS. International Covenant on Civil and Political Rights.

[59] UNITED NATIONS. International Covenant on Economic, Social and Cultural Rights.

[60] UNITED NATIONS. The Universal Declaration of Human Rights.

[61] VAN HORENBEECK, M. Crouching PowerPoint, Hidden Trojan. In *24th Chaos Communications Congress* (2007).

[62] VAN HORENBEECK, M. Cyber attacks against Tibetan communities. Tech. rep., Sans Institute, 2008.

[63] VAN HORENBEECK, M. Is Troy Burning? An overview of targeted trojan attacks. In *SANSFire 2008* (2008).

[64] VILLENEUVE, N. Human Rights and Malware Attacks. Tech. rep., Citizen Lab, University of Toronto, 2010.

[65] VILLENEUVE, N. Nobel Peace Prize, Amnesty HK and Malware. Tech. rep., Citizen Lab, University of Toronto, 2010.

[66] VILLENEUVE, N. Trends in targeted attacks. Tech. rep., Trend Micro, 2011.

[67] VILLENEUVE, N., AND WALTON, G. Targeted Malware Attack on Foreign Correspondents based in China. Tech. rep., Information Warfare Monitor, University of Toronto, 2009.

[68] VILLENEUVE, N., AND WALTON, G. 0day: Civil Society and Cyber Security. Tech. rep., Information Warfare Monitor, University of Toronto, 2009.

[69] XIANG, G., HONG, J., ROSE, C. P., AND CRANOR, L. CANTINA+: A Feature-Rich Machine Learning Framework for Detecting Phishing Web Sites. *ACM Trans. Inf. Syst. Secur. 14*, 2 (Sept. 2011), 21:1—-21:28.

[70] ZHANG, L., ZHU, J., AND YAO, T. An Evaluation of Statistical Spam Filtering Techniques. *Transactions on Asian Language Information Processing 3*, 4 (Dec. 2004), 243–269.

[71] ZHOU, Y., MULEKAR, M. S., AND NERELLAPALLI, P. Adaptive Spam Filtering Using Dynamic Feature Space. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence* (Nov. 2005), IEEE Computer Society, pp. 302–309.

```
From:  world fdc <fdc2008paris@gmail.com>
To:  [Tibet Group 1]
Subject:  Invitation

Please reply

1 Attachment:  invitation.doc
```

Figure 6: Example of e-mail with Targeting Score 1

```
From:  ciran nima <nimaciran@gmail.com>
To:  [Tibet Group 1]
Date:  18 Aug 2011
Subject:  Truth of monk dies after setting
himself on fire

Truth of monk dies after setting himself on
fire

1 Attachment:  Truth of monk dies after
setting himself on fire.doc
```

Figure 7: Example of e-mail with Targeting Score 2

Notes

[1] We report on results from other collection sources (e.g. NIDS alerts, website monitoring, and interviews), and cluster analysis of campaigns in a forthcoming technical report available at https://citizenlab/targeted-threats

Appendix

A Examples of targeted e-mails

In this section, we provide specific examples of e-mails that would be assigned targeting scores described in Section 4.2.2.

Targeting Score 1 (Targeted Not customized). The e-mail in Figure 6 was sent to Tibet group 1. The message content and sender are vague and do not relate to the interest of the group. The attachment is a word document implanted with malware. The lack of relevant information in this message gives it a score of 1 (targeted, not customized).

Targeting Score 2 (Targeted, Poorly Customized). The e-mail in Figure 7 was sent to Tibet group 1. It references Tibetan self-immolations which is an issue of interest to the group. However, the sender does not appear to be from a real person or organization. The message content is terse and does not referenced information that can be externally validated. Therefore this message scores a 2 (targeted, poorly customized).

```
From:  Palden Sangpo
<palden.sangpo@tibetancareers.org>
Subject:  Activity Report from Tibetan
Career Centre, Bylakuppe
Date:  24 Jan 2013
To:  [Tibet Group 2]

Dear Sir/Madam,

Tashi Delek.

Please find the attachment of the activity
report of Tibetan Career Centre, Bylakuppe
with this mail.  As I was asked to send
this activity report to your office.

Thank you.

Regards,
Palden Sangpo, Consultant.
Tibetan Career Centre,
Old Guest House, Lugsam Tibetan Settlement
Office,
PO Bylakuppe, Mysore District, Karnataka
State - 571 104
E-mail:  palden.sangpo@tibetancareers.org,
MO +91 9901407808, Off +91 8971551644
www.tibet.jobeestan.com

1 Attachment:  Report to CTA home.doc
```

Figure 8: Example of e-mail with Targeting Score 3

```
From:  Cheng Li <chengli.brookings@aol.com>
Subject:Happy Tib Losar and Ask You a
Favour
23 Feb 2012
To:  [Tibet Group 1]

Dear [Redacted]

I am Cheng Li from John L. Thornton China
Center of Brookings.  I will attend an
annual meeting on Religious Research
with CIIS in Shanghai next week, and
plan to take the chance to visit Tibet.
Attached is a list of tibetans who have
self-immolated from 2009 which my assistant
prepared for me, but i am not sure of its
accuracy.  Would you please have a look
and make necessary corrections.  I will be
really much appreciated if you could do me
the favor and offer some more information
about the latest happenings inside tibet.

Thank you again and happy Tib losar!

Cheng Li
Director of Research, John L. Thornton
China Center
Brookings Institution

1 Attachment:  list_of_self_immolations.
xls
```

Figure 9: Example of e-mail with Targeting Score 4

Targeting Score 3 (Targeted Customized). The e-mail in Figure 8 was sent to Tibet group 2. On the surface it appears to be a professional e-mail from "Palden Sangpo" a consultant at the Tibet Career Centre. The e-mail sender address and signature reference accurate contact details that can be easily verified through an Internet search. However, the e-mail headers reveal the purported e-mail sender address is fraudulent and the actual sender was `albano_kuqo@gmx.com`. The e-mail generally addresses the organization rather than the individual recipient. Therefore this message scores a 3 (targeted, customized).

Targeting Score 4 (Targeted Personalized). The e-mail in Figure 9 was sent to Tibet group 1. It is directly addressed to the director of the group and appears to come from Mr. Cheng Li, a prominent China scholar based at the Brookings Institute. The e-mail address is made to appear to be from Mr. Cheng Li, but from an AOL account (chengli.brookings@aol.com) that was registered by the attackers. The message asks the recipient for information on recent Tibetan self-immolations. The level of customization and personalization used in this message gives it a score of 4 (targeted, personalized).

Targeting Score 5 (Targeted Highly Personalized). Targeting scores of 5 (targeted, highly personalized) require reference to internal information to the target organization that could *not be* obtained through open sources. Examples of messages scoring at this level include an e-mail that purported to come from a funder of China Group 3 that provided details of an upcoming meeting the group actually had scheduled with the funder. In another example, Tibet Group 2 and Tibet Group 3 received separate e-mails that contained specific personal details about a South African group's visit to Dharamsala, India that appear to have been repurposed from a real private communication. The malicious attachment contained an authentic travel itinerary, which would be displayed after the user opened the document. The private information used in these messages suggest that the attackers performed significant reconnaissance of these groups and likely obtained the information through prior compromise.

A Look at Targeted Attacks Through the Lense of an NGO

Stevens Le Blond[1] Adina Uritesc[1] Cédric Gilbert[1]
Zheng Leong Chua[2] Prateek Saxena[2] Engin Kirda[3]

[1]*MPI-SWS* [2]*National Univ. of Singapore* [3]*Northeastern Univ.*

Abstract

We present an empirical analysis of targeted attacks against a human-rights Non-Governmental Organization (NGO) representing a minority living in China. In particular, we analyze the social engineering techniques, attack vectors, and malware employed in malicious emails received by two members of the NGO over a four-year period. We find that both the language and topic of the emails were highly tailored to the victims, and that sender impersonation was commonly used to lure them into opening malicious attachments. We also show that the majority of attacks employed malicious documents with recent but disclosed vulnerabilities that tend to evade common defenses. Finally, we find that the NGO received malware from different families and that over a quarter of the malware can be linked to entities that have been reported to engage in targeted attacks against political and industrial organizations, and Tibetan NGOs.

1 Introduction

In the last few years, a new class of cyber attacks has emerged that is more targeted at individuals and organizations. Unlike their opportunistic, large-scale counterparts, *targeted attacks* aim to compromise a handful of specific, high-value victims. These attacks have received substantial media attention, and have successfully compromised a wide range of targets including critical national infrastructures [19], Fortune 500 companies [23], news agencies [20], and political dissidents [10, 11, 16].

Despite the high stakes involved in these attacks, the ecosystem sustaining them remains poorly understood. The main reason for this lack of understanding is that victims rarely share the details of a high-profile compromise with the public, and they typically do not disclose what sensitive information has been lost to the attackers. According to folk wisdom, attackers carrying out targeted attacks are generally thought to be state-sponsored. Examples of national organizations that have been reported to be engaged in targeted attacks include the NSA's office of Tailored Access Operations (TAO) [3] and the People's Liberation Army's Unit 61398 [15]. Recently, researchers also attributed attacks in the Middle East to the governments of Bahrain, Syria, and the United Arab Emirates [16].

There now exists public evidence that virtually every computer system connected to the internet is susceptible to targeted attacks. The Stuxnet attack even successfully compromised air-gapped Iranian power plants [19] and was able to damage the centrifuges in the facility. More recently, Google, Facebook, the New York Times, and many other global companies have been compromised by targeted attacks. Furthermore, political dissidents and Non-Governmental Organizations (NGOs) are also being targeted [10, 11, 16].

In this paper, we analyze 1,493 suspicious emails collected over a four-year period by two members of the World Uyghur Congress (WUC), an NGO representing an ethnic group of over ten million individuals mainly living in China. WUC volunteers who suspected that they were being specifically targeted by malware shared the suspicious emails that they received with us for analysis. We find that these emails contain 1,176 malicious attachments and target 724 unique email addresses belonging to individuals affiliated with 108 different organizations. This result indicates that, despite their targeted content, these attacks were sent to several related victims (e.g., via Cc). Although the majority of these targeted organizations were NGOs, they also comprised a few high-profile targets such as the New York Times and US embassies.

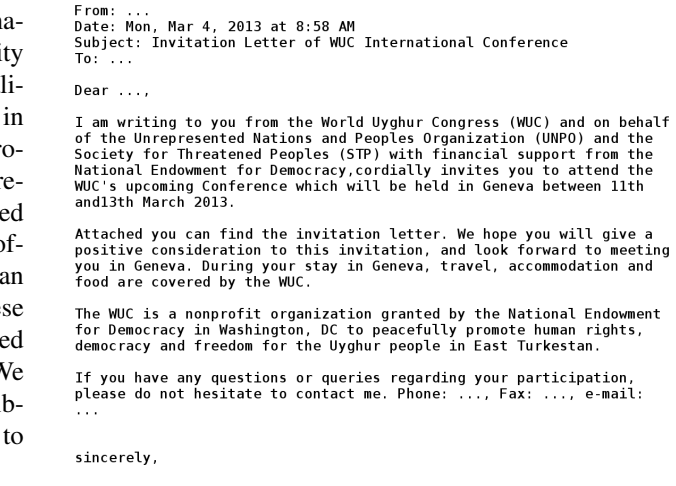
We leverage this dataset to perform an empirical analysis of targeted attacks in the wild. First, we analyze the engineering techniques and find that the language and topic of the malicious emails were tailored to the mother tongue and level of specialization of the victims. We also find that sender impersonation was common and that some attacks in our dataset originated from compromised email accounts belonging to high-profile ac-

tivists. Second, whereas recent studies report that malicious archives and executables represented the majority of the targeted-attack threat [15, 22], we find that malicious documents were the most common attack vector in our dataset. Although we do not find evidence of zero-day vulnerabilities, we observe that most attacks used recent vulnerabilities, that exploits were quickly replaced to adapt to new defense mechanisms, and that they often bypassed common defenses. Third, we perform an analysis of the first-stage malware delivered over these malicious emails and find that WUC has been targeted with different families of malware over the last year. We find that over a quarter of these malware samples exhibited similarities with those used by entities reported to have carried out targeted attacks.

Our work complements existing reports on targeted attacks such as GhostNet, Mandiant, and Symantec Internet Security Threat (ISTR) 2013 [11, 15, 22]. Whereas the GhostNet and Mandiant reports focus on the attack lifecycle *after* the initial compromise, this study provides an in-depth analysis of the reconnaissance performed *before* the compromise. We note that both approaches have pros and cons and are complementary: While it is hard for the authors of these reports to know *how* a system became compromised in retrospect, it is equally hard for us to know *if* the observed attacks will compromise the targeted system(s). Finally, whereas ISTR provides some numbers about reconnaissance analysis for industrial-espionage attacks [22], we present a thorough and rigorous analysis of the attacks in our dataset.

Finally, to foster research in this area, we release our dataset of targeted malware to the community [4].

Scope. Measuring real-world targeted attacks is challenging and this paper has a number of important biases. First, our dataset contains mainly attacks against the Uyghur and human-rights communities. While the specifics of the social engineering techniques (e.g., use of Uyghur language) will vary from one targeted community to another, we argue that identifying commonly used techniques (e.g., topic, language, senders' impersonation) and their purpose is a necessary step towards designing effective defenses. Another limitation of our dataset is that it captures only targeted attacks carried out over email channels and that were detected by our volunteers. Although malicious emails seem to constitute the majority of targeted attacks, different attack vectors such as targeted drive-by downloads are equally important. Finally, we reiterate that the goal of this study is to understand the reconnaissance phase occurring *before* a compromise. Analyzing second-stage malware, monitoring compromised systems, and determining the purpose of targeted attacks are all outside of the scope of this paper and are the topic of recent related work [10, 16]. We discuss open research challenges in Section 6.



```
From: ...
Date: Mon, Mar 4, 2013 at 8:58 AM
Subject: Invitation Letter of WUC International Conference
To: ...

Dear ...,

I am writing to you from the World Uyghur Congress (WUC) and on behalf
of the Unrepresented Nations and Peoples Organization (UNPO) and the
Society for Threatened Peoples (STP) with financial support from the
National Endowment for Democracy,cordially invites you to attend the
WUC's upcoming Conference which will be held in Geneva between 11th
and13th March 2013.

Attached you can find the invitation letter. We hope you will give a
positive consideration to this invitation, and look forward to meeting
you in Geneva. During your stay in Geneva, travel, accommodation and
food are covered by the WUC.

The WUC is a nonprofit organization granted by the National Endowment
for Democracy in Washington, DC to peacefully promote human rights,
democracy and freedom for the Uyghur people in East Turkestan.

If you have any questions or queries regarding your participation,
please do not hesitate to contact me. Phone: ..., Fax: ..., e-mail:
...


sincerely,
```

Figure 1: Screenshot of a malicious email with an impersonated sender, and a malicious document exploiting Common Vulnerabilities and Exposures (CVE) number 2012-0158 and containing malware. **The email replays an actual announcement about a conference in Geneva and was edited by the attacker to add that all fees would be covered.**

2 Overview

Context. WUC, the NGO from which we have received our dataset, represents the Uyghurs, an ethnic minority concentrated in the Xinjiang region in China. Xinjiang is the largest Chinese administrative division, has abundant natural resources such as oil, and is China's largest natural gas-producing region. WUC frequently engages in advocacy and meeting with politicians and diplomats at the EU and UN, as well as collaborating with a variety of NGOs. Rebiya Kadeer, WUC's current president, was the fifth richest person in China before her imprisonment for dissent in 1996, and is now in exile in the US. Finally, WUC is partly funded by the National Endowment for Democracy (NED), a US NGO itself funded by the US Congress to promote democracy. (We will see below that NED has been targeted with the same malware as WUC.)

WUC has been a regular target of Distributed Denial of Service (DDoS) attacks and telephone disruptions, as well as targeted attacks. For example, the WUC's website became inaccessible from June 28 to July 10, 2011 due to such a DDoS attack. Concurrently to this attack, the professional and private phone lines of WUC employees were flooded with incoming calls, and the WUC's contact email address received 15,000 spam emails in one week.

Data acquisition. In addition to these intermittent threats, WUC employees constantly receive suspicious emails impersonating their colleagues and containing

malicious links and attachments. These emails consistently evade spam and malware defenses deployed by webmail providers and are often relevant to WUC's activities. In fact, our volunteers claim that the emails are often so targeted that they need to confirm their legitimacy with the impersonated sender in person. For example, Figure 1 shows the screenshot of such an email that replays the actual announcement for a conference in Geneva organized by WUC. As a result, WUC members are wary of any emails containing links or attachments, and some of them save these emails for future inspection. We came in contact with two WUC employees who shared the suspicious emails that they had received (with consent from WUC). The authors of this work were not involved in the data collection.

Characteristics of the dataset. The two volunteers shared with us the headers and content of 1,493 suspicious emails that they received over a four-year period. 1,178 (79%) of these emails were sent to the private email addresses of the two NGO employees from whom we obtained the data, 16 via the public email address of the WUC, and the remaining 299 emails were forwarded to them (126 of these by colleagues at WUC). Overall, 89% of these emails were received directly by our volunteers or their colleagues at WUC. As we will see below, they also contain numerous email addresses in the To and Cc fields belonging to individuals that are not affiliated with WUC.

The emails contained 209 links and 1,649 attachments, including 1,176 with malware (247 RAR, 49 ZIP, 144 PDF, and 655 Microsoft Office files, and 81 files in other formats). Our analysis revealed 1,116 *malicious emails* containing malware attachments. (We were not able to verify the maliciousness of the links as most of them were invalid by the time we obtained the data.) In the following, we analyze malicious emails exclusively and we refer to *malicious archives or documents* depending on whether they contained RAR or ZIP, PDF or Microsoft Office documents, respectively. Finally, the volunteers labeled the data wherever necessary, enabling us, for example, to establish that the sender of the emails was impersonated for 84% of the emails. Table 1 summarizes the main characteristics of these malicious emails.

Scope of the dataset. Analyzing the headers of the malicious emails revealed a surprisingly large number of recipients in the To or Cc fields. In particular, we observed that malicious emails had been sent to 1,250 unique email addresses and 157 organizations. A potential explanation for this behavior could be that the attacker tampered with the email headers (e.g., via a compromised SMTP server) as part of social engineering so these emails were only delivered to our volunteers, despite the additional indicated recipients. To test this hypothesis, we considered only those emails received directly by our volunteers, originating from well-known webmail domains (i.e., aol.com, gmx.de, gmx.com, gmail.com, googlemail.com, hotmail.com, outlook.com, and yahoo.com), and verified via Sender Policy Framework (SPF) *and* DomainKeys Identified Mail (DKIM). SPF and DKIM are methods commonly used to authenticate the sending server of an email message. By verifying that these malicious emails originated from well-known webmail servers, we obtain 568 malicious emails whose headers are very unlikely to have been tampered with by the attacker. By repeating our above analysis on these emails only, we obtain 724 unique email addresses and 108 organizations. Other organizations besides WUC include NED (WUC's main source of funding and itself funded by the US congress), the New York Times, and US embassies. In summary, while we obtained our dataset from two volunteers working for a single organization, it offers substantial coverage not only of one NGO, but also of those attacks against multiple NGOs in which attackers target more than one organization with the same email. We show the full list of organizations targeted in our dataset in Appendix A.

What are targeted attacks? There is no precise definition of targeted attacks. In this paper, we loosely define these attacks as *low-volume*, *socially engineered* communication which entices *specific* victims into installing malware. In the dataset we analyze here, the communication is by email, and the mechanism of exploitation is primarily using malicious archives or documents. A targeted victim, in this work, refers to specific individuals, or an organization as a whole. When necessary, we also use the term volunteer(s) to distinguish between our two collaborators and other victims.

The terms targeted attacks and Advanced Persistent Threats (or APTs) are often used interchangeably. As this paper focuses on the reconnaissance phase of targeted attacks (occurring before a compromise), we cannot measure how long attackers would have remained in control of the targeted systems (i.e., their persistency). As a result, we simply refer to these attacks as targeted attacks, and not APTs, throughout the rest of this paper. We discuss specific social engineering characteristics that make targeted attacks difficult to detect by unsuspecting average users in Section 3, the attack vectors used in our dataset in Section 4, and the malware families they install in Section 5. Finally, we will discuss open research challenges in Section 6.

Ethics. The dataset was collected prior to our contacting WUC and for the purpose of future security analysis. Furthermore, WUC approved the disclosure of all the information contained in this paper and requested that the organization's name not be anonymized.

Table 1: Summary of our dataset originating from two volunteers. *Malicious* indicates the fraction of emails containing malware, *Impersonated* the fraction of emails with an impersonated sender, *# recipients* and *# orgs* the number of unique email addresses that were listed in the To and Cc fields of the malicious emails and the corresponding number of organizations, respectively.

	Beginning - end	*Size*	*Malicious*	*Impersonated*	*# recipients*	*# orgs*
1st volunteer	Sept 2012 - Sept 2013	98 MB	154/241 (64%)	141/154 (92%)	124	25
2nd volunteer	Sept 2009 - Jul 2013	818 MB	962/1,252 (77%)	802/962 (83%)	666	102
Total	Sept 2009 - Sept 2013	916 MB	1,116/1,493 (75%)	943/1,116 (84%)	724	108

3 Analysis of social engineering

The GhostNet, Mandiant, ISTR, and other reports [11, 15, 22] mention the use of socially-engineered emails to lure their victims into installing malware, clicking on malicious links, or opening malicious documents. For example, the GhostNet report refers to one spoofed email containing a malicious DOC attachment, and the Mandiant report to one email sent from a webmail account bearing the name of the company's CEO enticing several employees to open malware contained in a ZIP archive. Concurrent work reports the use of careful social engineering against civilians and NGOs in the Middle East [16] and also Tibetan and human-rights NGOs [10]. Despite this anecdotal evidence, we are not aware of any rigorous and thorough analysis of the social engineering techniques employed in targeted attacks. In this section, we seek to answer the following questions in the context of our dataset:

- *What social traits of victims are generally exploited?* Do attackers generally impersonate a sender known to the victim and if so who do they choose to impersonate?
- *Who are the victims?* Are malicious emails sent only to specific individuals, to entire organizations, or communities of users?
- *When are users being targeted?* When do users start being targeted? Are the same users frequently being targeted and for how long? Are several users from the same organization being targeted simultaneously?

3.1 Methodology

The analysis below focuses on 1,116 malicious emails received between 2009 and 2013.

Topics and language. To attempt to understand how well the attacker knows his victims, we manually categorized the emails (coded) by topic and language. (Unless indicated otherwise, the analysis below was performed on emails that were coded by one of the author.) The topic was determined by reading the emails' titles and bodies and, in cases where emails were not written in English, we also used an online translation service. Emails whose topic was still unclear after using the translator were labeled as *Unknown*.

Targeted victims. To determine the targeted victims of these attacks, we searched the email addresses and full names of the senders and receivers for the malicious emails originating from trustworthy SMTP servers. When available, we used their public profiles available on social media websites such as Google, Facebook, and Skype to determine their professional positions and organizations. We assume we have found the social profile of a victim if one of the three following rules applies (in that order): First, if the social profile refers directly to the email address seen in the malicious email; second, if the social profile refers to an organization whose domain matches the victims' email address; or third, if we find contextual evidence that the social profile is linked to WUC, Uyghurs, or the topic of the malicious email. Out of 724 victims' email addresses, we found the profile of 32% (237), 4% (30), and 23% (167) using the first, second, and last rule, respectively.

Organizations and industries. In the following, *WUC* refers to victims directly affiliated with the organization (including our volunteers). Other *Uyghur NGOs* include Australia, Belgium, Canada, Finland, France, Japan, Netherlands, Norway, Sweden, and UK associations. *Other NGOs* include non-profit organizations such as Amnesty International, Reporters Without Borders, and Tibetan NGOs. *Academia*, *Politics*, and *Business* contain victims working in these industries. Finally, *Unknown* corresponds to victims for which we were not able to determine an affiliation.

Ranks. We also translated the professional positions of the victims into one of the three categories: *High*, *Medium*, and *Low* profile. We consider professional leadership positions such as chairpersons, presidents, and executives as high-profile, job positions such as assistants, and IT personnel as medium-profile, and unknown and shared email addresses (e.g., NGO's contact information) as low-profile.

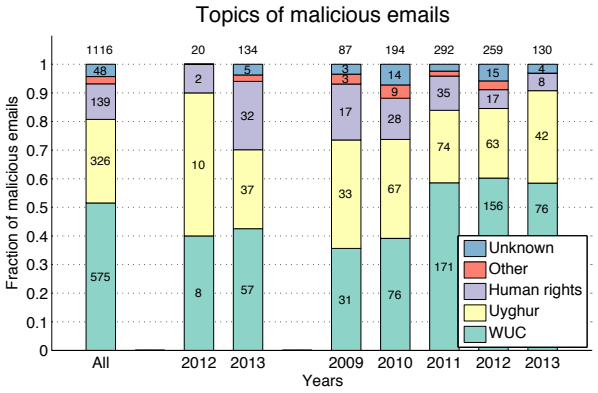


Figure 2: Distribution of the topics of the malicious emails for each year of the dataset shared by our two volunteers. The left bar corresponds to the data shared by both volunteers, and the next two bar groups to each year of the data shared by our first and second volunteer, respectively. **The content of malicious emails is targeted to the victims.**

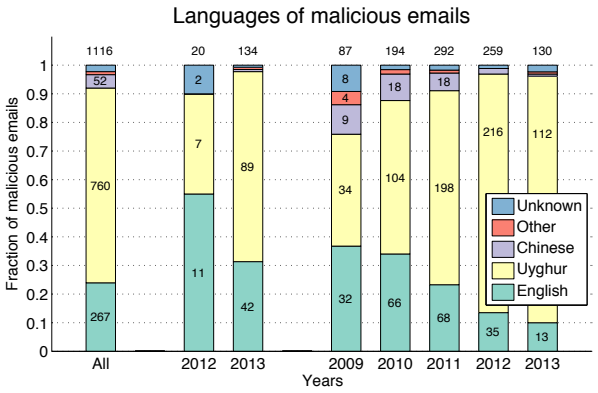


Figure 3: Distribution of languages for each year of our dataset. **Malicious emails employ the language of their victims.**

Impersonation. Finally, to understand the social context of the attack, each of our volunteers coded (based on her experience within the organization) all the email addresses of the senders into one of five categories: *Spoofed*, *Typo*, *Name*, *Suspicious*, or *Unknown*. (Coding was done based exclusively on the personal knowledge of the volunteers.) An email is marked as *Spoofed* if it bears the exact sender email address of a person known to our volunteers, as *Typo* if it resembles a sender email address known to the receiver but is not identical, and as *Name* if the attacker used the full name of a volunteer's contact (with a different email address). Finally, email addresses that look as if they had been generated by a computer program (e.g., uiow839djs93j@yahoo.com) are labeled as *Suspicious* and all remaining emails as *Unknown*. Our assumption is that, because our volunteers received most of the malicious emails directly, they were likely to recognize cases where their contacts were being impersonated. We note that labeling is conservative: Our volunteers may sometimes label *Spoofed* or *Typo* addresses as *Unknown* because they do not know the person impersonated in the attack. This may happen, for example, in cases where they were not the primary target of the attack (e.g., they appeared in Cc).

Limitations. Our dataset originates from WUC and is limited to those victims that were targeted together with that organization. We will see that these victims were often NGOs. As a result, the social engineering techniques observed here may differ from attacks against different entities such as companies, political institutions, or even other NGOs. Despite these limitations, we argue that this analysis is an important first step towards understanding the human factors exploited by targeted attacks.

3.2 Results

In this subsection, we discuss the results of our analysis of the social engineering techniques used in the malicious emails.

Topics and language. The topic of malicious emails in our dataset can generally be classified into one of three categories: WUC, Uyghur, and human-rights. In particular, we observed 51% (575) of malicious emails pertaining to WUC, 29% (326) to Uyghurs, 12% (139) to human-rights, and 3% (28) to other topics. In addition, the native language of the victim is often used in the malicious emails. In fact, 69% (664) of the emails sent to the second volunteer were written in the Uyghur language, and 62% (96) for the first one. These results indicate that attackers invested significant effort to tailor the content of the malicious emails to their victims, as we see in Figure 2 and Figure 3.

Specialized events. In addition to being on topic, we also observed that emails often referred to specific events that would only be of interest to the targeted victims. Throughout our dataset, we found 46% of events (491) related to organizational events (e.g., conferences). We note that these references are generally much more specialized than those used in typical phishing and other profit-motivated attacks. For example, Figure 1 shows a screenshot of an attack that replayed the announcement of a conference on a very specialized topic. The malicious email was edited by the attacker to add that all fees would be covered (probably to raise the target's interest).

Impersonation. We find that attackers used carefully crafted email addresses to impersonate high-profile identities that the victims may directly know. That is, attackers used one of the following four techniques to add legitimacy to a malicious email: First, 41% (465) of the

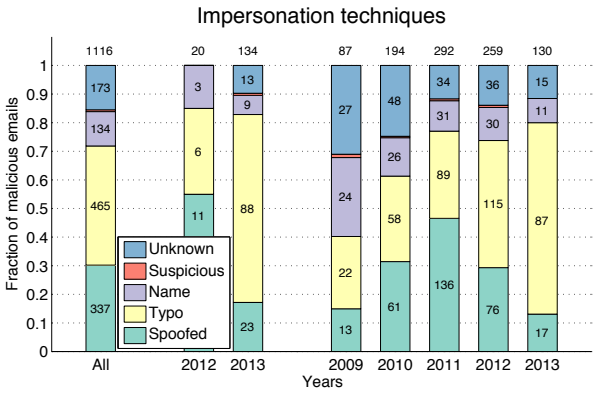


Figure 4: Distribution of senders' impersonation techniques for each year of our dataset. **Malicious emails spoof the email address of a contact of the volunteers, use a very similar address controlled by the attacker, or a contact's full name.**

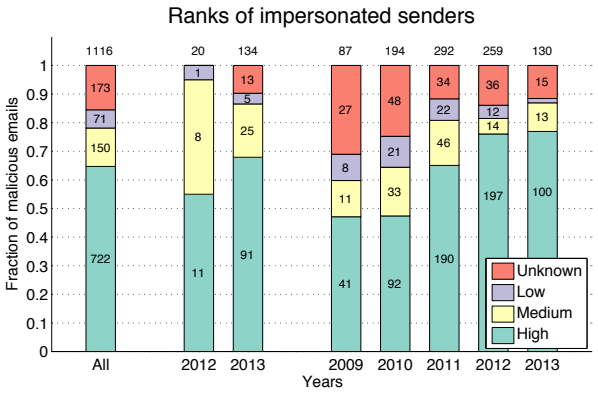


Figure 5: Distribution of impersonated senders' ranks for each year of our dataset. **Malicious emails often impersonate high-profile individuals.**

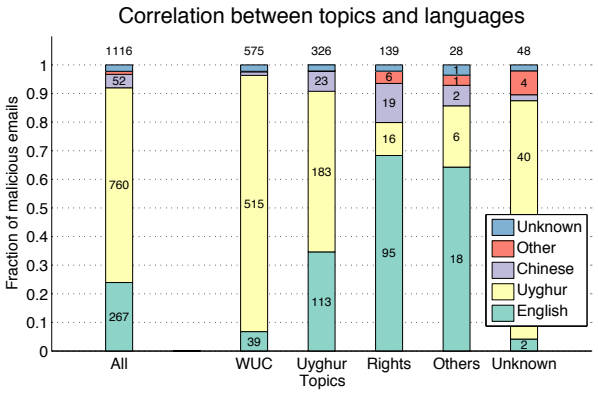


Figure 6: Distribution of languages employed to write about the main topics of malicious emails. **There is a strong correlation between malicious emails' topics and the language in which they are written.**

email addresses have *Typos* (i.e., the email address resembles known sender addresses, but with minor, subtle differences). These email addresses are identical to legitimate ones with the exception of a few characters being swapped, replaced, or added in the username. Second, 12% (134) of the senders' full names corresponded to existing contacts of the volunteers. Third, we find that most email addresses belonged to well-known email providers — Google being the most prominent with 58% of all emails using the Gmail or GoogleMail domains, followed by Yahoo with 16%.

Fourth, we find that 30% (337) of the sender emails were spoofed (i.e., the email was sent from the address of a person that the volunteer knows). This observation suggests that the attacker had knowledge of the victim's social context, and had either spoofed the email header, or compromised the corresponding email account. To identify a subset of compromised email accounts, we consider spoofed emails authenticated by the senders' domains using both SPF and DKIM. To reduce the chances of capturing compromised servers instead of compromised accounts, we also consider only well-known, trustworthy domains such as GMail. This procedure yields malicious emails that were likely sent from the legitimate account of the victims' contacts. We found that three email accounts belonging to prominent activists, including two out of 10 of the WUC leaders, were compromised and being used to send malicious emails. We have alerted these users and are currently working with them to deploy defenses and more comprehensive monitoring techniques, as we will discuss in Section 6.

We show the distributions of malicious emails sent with spoofed, typo, suspicious, or unknown email addresses in Figure 4, and the ranks of the impersonated senders in Figure 5. (We do not show the corresponding ranks for receivers because NGOs generally function with a handful of employees, all playing a key role in the organization.)

Targeted victims. For the analysis below, which leverages other recipients besides our two volunteers, we further filter emails to keep only those originating from well-known domains (as described in Section 2). Doing this leaves us with 568 malicious emails that are likely to have indeed been sent to all the email addresses in the header. We find that the attacks target more organizations than WUC, including 38 *Uyghur NGOs*, 28 *Other NGOs*, as well as 41 *Journalistic*, *Academic*, and *Political* organizations. (See Appendix A for the complete list of targeted organizations.) Interestingly, we find a strong correlation between the topic of an email and the language in which the email was written, as we show in Figure 6. Our results show that English was more and

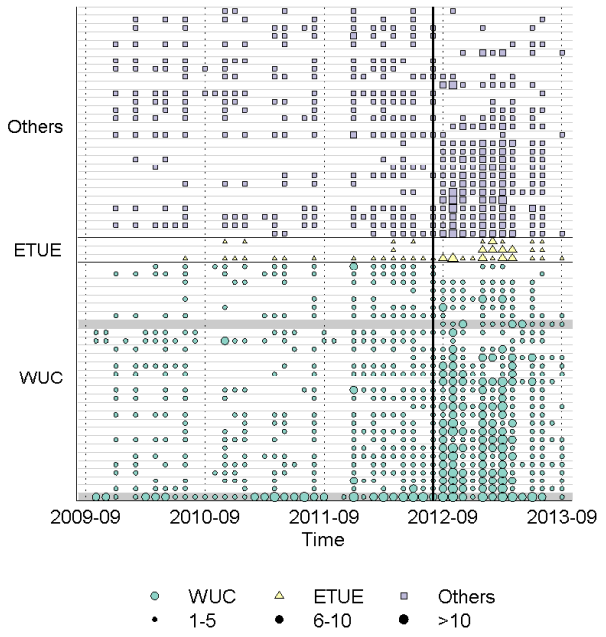


Figure 7: Timeline of attacks, in number of malicious emails per month, against the 60 most targeted victims (our two volunteers' rows are shaded and the vertical line corresponds to one of our volunteer joining WUC). The Y axis represents victims grouped by organization. *ETUE* corresponds to the East Turkestan Union in Europe NGO and *Others* to different organizations. **Each of the top 60 victims has been frequently attacked over the last four years and several victims from the same NGOs were attacked simultaneously.**

more common as the topic became less and less specialized. We hypothesize that attackers may have sent the same email messages to several recipients with similar interests to reduce the costs involved in manually crafting these emails.

Timing. Our dataset shows that the same victims were frequently targeted and that several members of the same organization were routinely targeted simultaneously. This suggests that attackers were using a "spray" strategy, trying to find the weakest links in the targeted organization, and hence, optimizing their chance of success. Spraying is clearly visible in Figure 7 where we see that the top 60 most targeted victims in our dataset received malicious emails often over the last four years. (We note that the dataset shared by one of our volunteers starts on August 2012, explaining why we observe more malicious emails after that date.) We also see that, 31 email accounts from individuals without affiliation to WUC were often targeted simultaneously to the WUC accounts.

Summary of Findings. We now revisit the initial questions posed at the beginning of this section. First, we saw that most emails in our dataset pertained to WUC, Uyghurs, or human-rights, were written in the recipient's mother tongue, and often referred to very specialized events. We also found that sender impersonation was common and that some email accounts belonging to WUC's leadership were compromised and used to spread targeted attacks. (We note that many more accounts may be compromised but remain dormant or do not appear as compromised in our dataset.) Second, we showed that numerous NGOs were being targeted simultaneously with WUC and that the specialization of emails varied depending on the recipient(s). Finally, we observed that the most targeted victims received several malicious emails every month and that attacks were sprayed over several organizations' employees.

4 Analysis of attack vectors

We now analyze the techniques used to execute arbitrary code on the victim's computer. The related work reports the use of malicious links, email attachments, and IP tracking services [10, 16]. Whereas ISTR 2013 reports that EXE are largely used in targeted attacks, and the Mandiant report that ZIP is the predominant format that they have observed in the last several years, we find that these formats represent 0% and 4% (49) of malicious attachments in our dataset, respectively. Instead, we find RAR archives and malicious documents to be the most common attack vectors. Hypotheses that may explain these discrepancies with the Mandiant report include the tuning of attack vectors to adapt to the defenses mechanisms used by different populations of email users (e.g., NGOs vs. corporations); Mandiant's attacker (APT1), mainly using primitive attack vectors such as archives; and/or Mandiant having excluded more advanced attack vectors, such as documents, from its report. However, in the absence of empirical data on APT1's attack vectors, we cannot test these hypotheses. In this section, we perform a quantitative study of the attack vectors employed in our dataset, and also analyze their dynamics. We seek to answer the following questions:

- *What attack vectors are being employed against WUC?* Do they generally rely only on human failures or also on software vulnerabilities? Do they evolve in time and if so, how quickly do they adapt to new defense mechanisms?
- *What is the efficacy of existing countermeasures?* As all malicious documents in our dataset used well-known vulnerabilities, would commercial, state-of-the-art defenses have detected all of them?

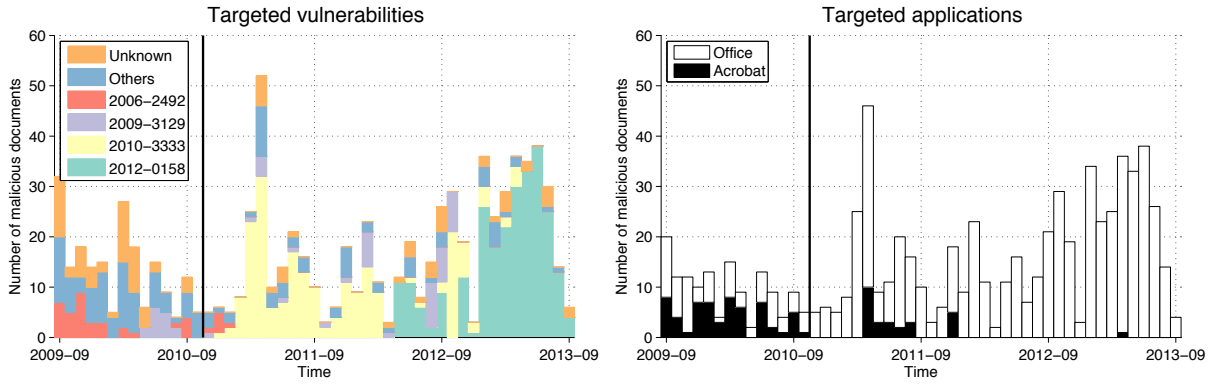


Figure 8: Number of malicious documents containing a given vulnerability (CVE) (left) and target application (right) for each month of our dataset. We represent the top four CVEs in number of attacks over the whole trace individually and *others* are represented in aggregate. The vertical line in November 2010 corresponds to the deployment of sandboxing in Acrobat Reader. **Although Acrobat Reader was the most targeted application until 2010, recent attacks mainly target the Office suite.**

4.1 Methodology

Malicious archives. To analyze the archives' contents, we extracted them in a disconnected VM environment and manually inspected their contents to determine the type of files they contain, independently of their extensions. In the case of EXE files, we also examined them manually to determine whether their Microsoft Windows icons were similar to those used for other file formats (e.g., JPEG) in order to persuade average users that they were not executable.

Malicious documents. We used two methodologies to determine the characteristics of the vulnerabilities being exploited by malicious documents. First, we submitted the documents to VirusTotal [1] for analysis. Each of the 45 Antivirus (AVs) on VirusTotal classified the checked sample as benign or malicious, and attached a "tag" describing the auxiliary information relating to the sample. Often the tag is a Common Vulnerabilities and Exposures (CVE) number, presumably corresponding to the signature that matched, but in some cases, the tag field is not a CVE; it is either tagged as "unknown" or contains a symptomatic description such as the inclusion of a suspicious OLE object. We refer to these three tags as *CVE*, *Unknown*, and *Heuristic*, respectively. Often all AVs reported a *Single* CVE but sometimes, they reported *Multiple*, conflicting CVEs. Once we collected all CVE tags, we then scraped the National Vulnerability Database [18] to obtain the release date and vulnerable applications for each of the CVEs that we found.

Second, we inspected the documents manually to confirm that they contain malware, and also used taint-assisted analysis both to verify the accuracy of the CVEs reported in AV reports and to investigate the presence of zero-day vulnerabilities.[1] The methodological details of our taint-assisted manual analysis are described in Appendix B.

Defenses. We performed a retrospective analysis of the protection offered by common defenses such as AV and webmail providers in the context of our malicious documents. For AV, we used VirusTotal to determine whether a malicious document is detected by the scanning engine of each AV, as described above. For webmail channels, we created an email account on GMail, Hotmail, and Yahoo, and used a dedicated SMTP server to send emails to that account with malicious documents attached. We considered malicious documents delivered without modifications as undetected by the webmail defenses. Otherwise, if an email or its attachment is dropped, or if the attachment's payload is modified, we considered it as detected. The analyses based on webmails and VirusTotal were performed in November 2013 and July 2014, respectively.

Limitations. As with social engineering, our analysis of attack vectors is biased towards NGOs. In addition, the above methodology is limited to the attack vectors captured in our dataset. For example, we miss attacks against the NGOs' web servers unless the corresponding malicious link appears in the suspicious emails.

Second, our taint-assisted analysis of vulnerabilities is limited to those documents for which we were able to analyze the logs manually. For example, we found that opening PDF files in our environment generated log files that were far too large (around 15GB in the median case) for manual analysis. As a result, we were able to manually confirm vulnerabilities only against Microsoft Office. However, despite this limitation, we were also able to determine which PDF documents contained malware through manual inspection.

[1] Hereafter, *zero-day vulnerabilities* refer to vulnerabilities that were not publicly disclosed at the time of the attack.

Table 2: List of well-known vulnerabilities exploited by malicious documents. *Release* corresponds to the release date of the vulnerability and *First* to its first exploitation in our data set (in number of days relative to the release date). *Resolved* corresponds to the number of Microsoft Office vulnerabilities that were mistagged in AV reports but that we were able to resolve using taint-assisted manual analysis.

CVE	*Release*	*First*	*Apps*	*# emails*	*Resolved*
2006-0022	06/13/06	1,191	Office	2	0
2006-2389	07/11/06	1,166	Office	18	16
2006-2492	05/20/06	1,125	Office	59	47
2007-5659	02/12/08	588	Acrobat	3	0
2008-0081	01/16/08	651	Office	1	0
2008-0118	03/11/08	1,010	Office	1	0
2008-4841	12/10/08	824	Office	1	0
2009-0557	06/10/09	405	Office	2	0
2009-0563	06/10/09	880	Office	31	0
2009-0927	03/19/09	180	Acrobat	11	0
2009-1862	07/23/09	68	Acrobat	3	0
2009-3129	11/11/09	188	Office	58	4
2009-4324	12/15/09	4	Acrobat	15	0
2010-0188	02/22/10	28	Acrobat	15	0
2010-1297	06/08/10	0	Acrobat	9	0
2010-2883	09/09/10	7	Acrobat	7	0
2010-3333	11/10/10	49	Office	220	0
2010-3654	10/29/10	0	Office	7	0
2011-0611	04/13/11	0	Acrobat	19	0
2011-0097	04/13/11	224	Office	3	0
2011-2462	12/07/11	2	Acrobat	5	0
2012-0158	04/10/12	37	Office	278	12
2013-0640	02/14/13	68	Acrobat	1	0

Finally, our defense analysis was performed in bulk, after the time of the attacks. As a result of the difference between the times of attack and analysis (up to four years for the first malicious documents), the detection rates reported hereafter should be treated as upper bounds. This is because the AV signatures at the time of the analysis were more up-to-date than they would have been at the time of the attack.

4.2 Results: Attack vectors

4.2.1 Malicious archives

We observed numerous targeted attacks leveraging social engineering and human failure to install malware on the victim's computer. In particular, we found 247 RAR and 49 ZIP containing malicious EXE. In 10 cases, the malicious archives were password protected with the password included in the email's body. We hypothesize that archiving was used as a rudimentary form of packer for the malware to evade detection by the distribution channels. Finally, we found that 20% of all EXEs contained in the archives used an icon that resembled a non-EXE, i.e., a DOC, JPEG, or PDF icon, in 20%, 19%, and 7% of the cases.

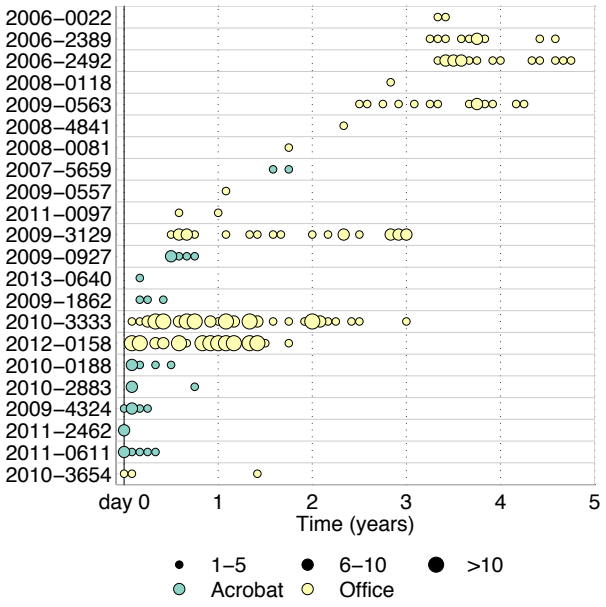


Figure 9: Timeline of the target vulnerabilities. The Y axis corresponds to CVEs and each circle to the number of CVEs seen each month after the public disclosure of the vulnerability (day 0). **All vulnerabilities were first targeted after their public disclosure.**

4.2.2 Malicious documents

We used taint-assisted analysis to resolve the conflicts due to AV mistagging and summarize the CVE information in Table 2. The number of conflicts resolved using taint-assisted manual analysis is reported in the last column *Resolved*. Additional taint-analysis results are reported in Appendix B.

Zero-day versus unpatched vulnerabilities. We find no evidence of the use of zero-day vulnerabilities against our dataset, but several uses of disclosed vulnerabilities within the same week as their public release date. In addition, we see in Figure 9 that vulnerabilities continued to be exploited for years after their disclosure, and this confirms that unpatched vulnerabilities represent a large fraction of attacks in our dataset. To ascertain the CVE being exploited in each sample, we used a combination of the telemetry data available in CVE tags generated by AVs, and a manual analysis to resolve cases where the tag was ambiguous. For each sample, we then recover the public disclosure date for the vulnerability manually, and treat it as the corresponding day-zero. By comparing the time of use in our email dataset, we are able to ascertain the lifetime of vulnerability exploits.

We find several instances of exploits that were used in publicly-reported targeted attacks in our dataset. For

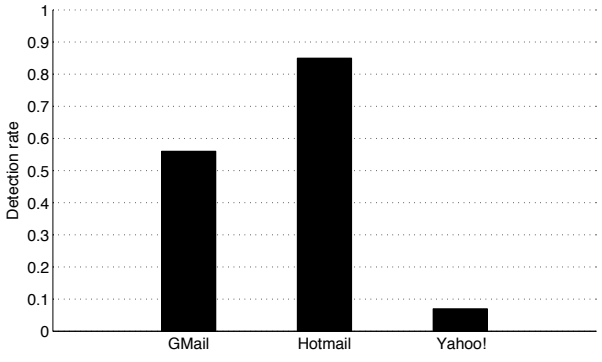


Figure 10: Detection rates of popular webmails for the malicious documents. **The efficacy of webmails to detect malicious documents varies widely.**

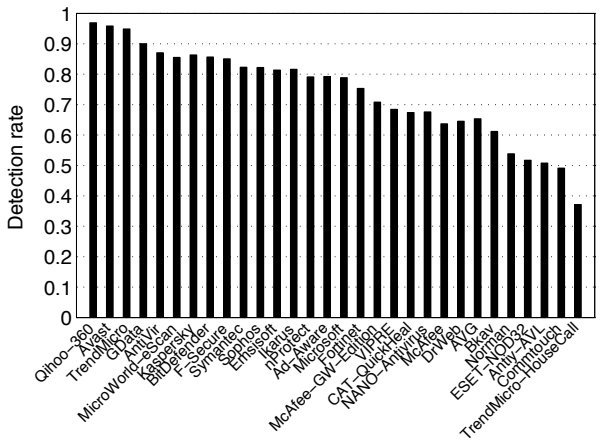


Figure 11: Detection rates of malicious documents for each of the top 30 AVs as reported by VirusTotal. **No single AV detected all malicious documents despite their use of well-known vulnerabilities.**

instance, vulnerabilities such as CVE-2009-4324, CVE-2010-3654, and CVE-2010-2883 have been reported to be zero-day vulnerabilities [6]. However, in our dataset, these vulnerabilities were used after their disclosure.

Evolution of target applications. Our data shows a sudden switch from Adobe Reader to Microsoft Office suite as the primary targeted application as of November 2010, as seen in Figure 8. We find a correlation between the time of this switch and two events: (a) the deployment of sandboxing defenses in Adobe Reader and (b) the disclosure of vulnerabilities in the Office suite. The first version of Acrobat Reader to support sandboxing for Windows (version 10.0) was released on November 15, 2010. Within the same month, a stack buffer overflow against Microsoft Office was released publicly (November 2010), reported as CVE-2010-3333. We see this CVE being massively exploited in our dataset as of January 2011, which is a time lag of two months. We observe the use of CVE-2010-3333 being replaced with CVE-2012-0158 in January 2013. This evidence suggests that attackers adapted their targeted vectors to use newly disclosed vulnerabilities within a few days to a few months of disclosure, and that updates to the security design of software reduces its exploitability in the wild (as one would expect).

4.3 Results: Bypassing common defenses

We now investigate the efficacy of existing defenses against malicious documents.

Email / Webmail Filtering. Despite the retrospective analysis of the malicious documents, we find that the detection rates of malicious documents for GMail, Hotmail, and Yahoo were still relatively low (see Figure 10). We also find that GMail failed to detect most malicious documents sent after March 2012. In particular, while the detection of documents sent before March 2012 was 73%, it is 28% after that date. Interestingly, 71% of the true positives for GMail after March 2012 corresponded to RTF files with all \r\n character sequences substituted with the \n character. While this substitution did not deactivate the malware, we observed that it broke the shellcodes embedded into these documents as they require the document size to remain unchanged to function properly. As a result, the malware was never executed. Although we cannot verify the purpose of this substitution, we note that its appearance coincided with that of the malicious RTF files. We conclude this discussion by pointing out that Yahoo's low detection rate is interesting as it claims to be using Symantec AV for its webmail service [12] — which, as we will see below, has a much higher detection rate.

Signature-based AV Scanning. In Figure 11, we show the detection rates for the top 30 VirusTotal AVs, sorted by decreasing detection rate of the malicious documents. There are two main takeaways from this graph. First, no single vendor detected all original malicious documents, even though we have seen that they used well-known vulnerabilities. For example, Qihoo, the vendor with the overall best efficacy, was unable to detect 3% of the malicious documents based on scanning. Second, we observe large variations among the efficacy of different AV vendors. That is, the detection rate dropped by 30% from the first to the twentieth AV (CAT QuickHeal) and the 15 AVs with the lowest detection rate (not shown) all had a detection rate of less than 35%.

Summary of Findings. We found that malicious documents are the most popular attack vectors in our dataset followed by malicious archives. Malicious documents tended to use newly released vulnerabilities, often within

Table 3: Summary of the malware clusters. For each cluster, we show the malware family (or an ID if we could not determine it), the number of malicious emails containing the malware, the number of Command and Control (C2) servers, the similarities in terms of communication protocols and C2 with malware attributed to known entities (*entity(Com,C2)*). **Our dataset contains several families of first-stage malware previously seen in targeted attacks carried out in the wild.**

Clusters	*1*	*2*	*Surtr*	*4*	*5*	*6*	*7*	*8*	*9*	*TravNet*
# samples	67 (9%)	58 (8%)	51 (7%)	37 (5%)	30 (4%)	22 (3%)	19 (2%)	19 (2%)	18 (2%)	13 (2%)
# C2	6	2	6	18	13	3	8	13	9	4
Similarity	DTL(C2)	—	DTL(C2)	—	—	—	—	—	—	TravNet(Com,C2)

a week, continued to utilize them for several years, and most of them used well-known instead of zero-day vulnerabilities. In particular, our taint-assisted manual analysis of Office documents did not reveal a single zero-day vulnerability in our dataset. This raises the question of whether defense mechanisms deployed in webmails and state-of-the-art commercial defenses are effective in blocking these well-known attacks. Furthermore, we found that malicious archives often contained EXE files that masquerade as pictures or documents.

5 Malware analysis

We now analyze the first-stage malware found in malicious documents. Unlike the Mandiant report, which provides an analysis for malware that targets different organizations and that (they claim) originates from the same group, our analysis focuses on all malware (in our dataset) that has targeted a single organization. By looking at targeted attacks from the perspective of the target rather than the attacker, our analysis enables us to determine whether WUC has been targeted with the same or different malware over the years. We also take a different approach from the authors of the GhostNet report who performed malware analysis on a few compromised systems belonging to different but related organizations. We instead analyze over six hundred malware samples used to establish a foothold on the targeted systems of a single organization. Our analysis differs from the related work in its scale and context [16] or focus [10]. This section aims to answer the following question:

- *Is WUC targeted with the same or different malware?* In the latter case, are there similarities between this first-stage malware and others found in targeted attacks in the wild?

5.1 Methodology

Our analysis below was done on 689 malware samples that we extracted from malicious documents.

Clustering. To make our analysis tractable for 689 malware samples, we started by clustering the malware based on its behavior. To do so, we ran the malicious EXE and DLL files in a disconnected sandboxed environment and hooked the function calls to resolve domain names and establish network communications. In addition, to obtain the TCP port number on which communication is done, we intercepted function calls to `gethostbyname` and returned a dummy routable IP address. As a result, the malware subsequently reveals the port number when it initiates a connection with the returned IP. (See Appendix C for the complete list of Command and Control (C2) domains.) Finally, we generated behavioral profiles for 586 samples, clustered them using an approach similar to [5, 14], and manually verified the accuracy of the resulting clusters.

Malware family and similarities. Similarly to Bailey et al. [5], we found that determining the malware family using AV signature scanning was unproductive. To determine whether our malware shares similarities with other known targeted malware, we relied on several reports on targeted attacks [9, 13]. We extracted the C2 domains and, when available, additional information about the malware (e.g., hashes and behavior) from these reports. Finally, we correlated the domains, IP addresses, hashes, and behavioral profiles with those from the reports in order to find similarities between the different sets of malware. We performed this analysis in February 2014.

Limitations. Our behavioral analysis was performed in a disconnected environment and as a result, it is limited to the first stage of the malware behavior. Studying the behavior of additional payload that would be downloaded after the compromise is beyond the scope of this paper and will be the subject of future work.

5.2 Results

We now analyze the malware clusters and their similarities with other targeted malware found in the wild.

Cluster sizes. We find that 57% of our malware belonged to the ten largest clusters (we show additional information about these clusters in Table 3). In total, five clusters (two in the top ten) used at least one of dtl6.mooo.com, dtl.dnsd.me, or dtl.eatuo.com as their

C2 domains, indicating some operational link between them. In fact, at the time of analysis, these three domains resolved into the same IP address and the malware in each cluster connected to different ports of the same server. Despite these apparent similarities, however, manual analysis of the behavioral logs revealed that their logic differed from one another, explaining their assignment to different clusters. Combined, these five clusters represented 24% of the malware that we analyzed.

Malware family and similarities. We found various degrees of similarities between our clusters and targeted attacks reported in the wild. First, the five clusters above had the same C2 as the DTL group reported by FireEye in November 2013 and that the malware was of the same family as one of these clusters' (*APT.9002,* not shown) [9]. In particular, we found that one of our samples in that cluster had the same MD5 hash as those described in the FireEye report and that eight had identical manifest resources. FireEye claims that this malware has been used in targeted attacks against various governmental and industrial organizations.

Second, malware in the *Surtr* cluster had the same behavioral profile as samples used against the Tibetan community in March 2012 [7]. Although the two sets of samples had different MD5 hashes, they both connected to the same C2 server (shared with *APT.9002*) on the same port number, and exhibited the same behavior to establish persistency on the victim's machine.

Third, our 13 TravNet samples exhibited similar behavior as those used against Indian targets in 2013 [2]. To do so, we obtained the samples used in India, generated their behavioral profiles, and compared them manually with the malware in our *TravNet* cluster. Although both sets connected to different C2 servers and exhibited variations in the way they searched the victims' file system, we found that they both used the same communication protocol with the C2.

Fourth, samples in another cluster communicated with the same C2 server and exhibited the same behavior as a Vidgrab sample found in a malicious document sent to a victim in Hong Kong in August 2013 [8].

Summary of findings. We found that WUC has been targeted with several malware families in the last year. We also showed that the *Surtr* and *APT.9002* clusters corresponded to malware that Citizenlab and FireEye identified as having targeted the Tibetan community, as well as other political and industrial organizations [7, 9]. Furthermore, 24% of our malware (including *Surtr*, *APT.9002* and three other clusters) had at least one C2 domain in common, which was identical to those of the Citizenlab and FireEye reports.

6 Future Work

Several directions for future work arise from this work. We briefly discuss them below.

Attack vectors and generalization. Our analysis is limited to attack vectors used against WUC. Similar studies on a wider range of targets would benefit understanding this emerging threat better. Further, our attack vectors distributed over email channels and have two main limitations. First, it is possible that our volunteers have been attacked via other channels besides email. Second, although we have seen various organizations targeted with the same malware as WUC, it is generally hard to determine with certainty which victims were the primary target of these attacks. Therefore, it is possible that other victims have been targeted with additional attack vectors when the attacks did not involve WUC. Further research is needed to overcome these limitations

Exploring different channels that attackers use for distributing malicious payloads is important. As a step towards this goal, we are currently collaborating with the Safebrowsing team at Google to investigate the emergent threat of watering-hole attacks. These attacks are conceptually very similar to drive-by download attacks with one key difference: They compromise *very specific* websites commonly visited by the targeted community (e.g., a company's website) and wait for victims to visit the website. As compared to spear phishing, watering-hole attacks offer the advantage of potentially targeting a fairly large number of victims (e.g., all employees of a large company) before raising suspicion. We conjecture that the small number of suspicious links in our dataset may be due to the small size of the targeted organizations and the public availability of their employees' email addresses.

Other attack vectors include but are not limited to packets injection to redirect victims to malicious servers (similar to those used in watering-hole attacks) and physical attacks on the victims' devices [3]. Detecting these attacks would require completely different methodologies than the one we used in this paper.

Monitoring. We have seen that a few high-profile members of the Uyghur community were compromised and that their email accounts were being used as stepping stones to carry out targeted attacks. Although it is possible that these email accounts were compromised via targeted attacks, we have not yet confirmed this hypothesis. More generally, we do not know yet what is the specific aim of these targeted attacks. Monitoring the full lifecycle of targeted attacks would require novel measurement systems, deployed at the end users, that can identify compromises without being detected.

Pinpointing the geolocation of attackers carrying out targeted attacks, or *attack attribution*, is another open

monitoring challenge. Marczak et al. were able to attribute targeted attacks to governments in the Middle East by analyzing relationships of cause and effect between compromises and real-world consequences [16]. In contrast to monitoring and attack attribution, this paper has presented a extensive, complementary analysis of the life cycle of targeted attacks *before* the compromise.

Large-scale malware analysis and clustering. We found it challenging to (a) cluster targeted malware and (b) locate similar samples. First, this malware sometimes exhibits significant similarity in its logic and different malware may also use the same Command and Control (C2) infrastructure. As a result, traditional clustering algorithms tend not to work very well. Second, we located similar samples based on a limited set of indicators such as C2, cryptographic hash, or YARA signatures, however, we feel that our current capability in that respect has a lot of room for improvement. We foresee that a search engine that can, for example, locate malware matching certain indicators out of an arbitrarily large corpus would be a useful instrument for researchers working on targeted attacks.

Our analysis of CVEs highlights that telemetry data from commercial AVs is not always reliable. Our analysis complemented with taint-analysis was largely manual and time-intensive. Analysis techniques to quickly diagnose known CVEs directly from given exploits is an open problem and perhaps one of independent interest.

Defenses. Our findings confirm that AVs may miss known CVEs, even years after their release dates. Clearly, known CVEs contribute a large part of the emerging threat of targeted attacks. Understanding why commercial AVs miss known attacks conclusively, for example to tradeoff false positives or performance for security, is an important research direction. Designing effective defenses against targeted attacks is a major research challenge which depends on our ability to understand the threat at hand. As part of future work, one could evaluate the effectiveness of novel defenses based on the findings from this paper. As a small step towards that goal, we plan to soon deploy a webmail plugin that combines metadata and stylometry analysis [17] to detect contact impersonation.

7 Conclusion

We have presented an empirical analysis of a dataset capturing four years of targeted attacks against a human-rights NGO. First, we showed that social engineering was an important component of targeted attacks with significant effort paid in crafting emails that look legitimate in terms of topics, languages, and senders. We also found that victims were targeted often, over the course of several years, and simultaneously with colleagues from the same organization. Second, we found that malicious documents with well-known vulnerabilities were the most common attack vectors in our dataset and that they tended to bypass common defenses deployed in webmails or users' computers. Finally, we provided an analysis of the targeted malware and showed that over a quarter of samples exhibited similarities with entities known to be involved in targeted attacks against a variety of industries. We hope that this paper, together with the public release of our malware dataset, will facilitate future research on targeted attacks and, ultimately, guide the deployment of effective defenses against this threat.

Acknowledgements. The authors would like to thank the anonymous reviewers, our shepherd Stuart Schechter, and Peter Druschel for their useful feedback. We would also like to acknowledge Karmina Aquino (F-Secure), Emiliano Martinez (VirusTotal), Mila Parkour (Contagio), and Nart Villeuneuve (FireEye) for their help locating similar malicious documents. Finally, we thank the Max Planck Society, the Ministry of Education of Singapore under Grant No. R-252-000-495-133, and the NSF under Grant No. CNS-1116777 for partially supporting this work.

References

[1] http://www.virustotal.com.

[2] Inside report APT attacks on indian cyber space. Tech. rep. http://g0s.org/wp-content/uploads/2013/downloads/Inside_Report_by_Infosec_Consortium.pdf.

[3] Inside TAO: Documents reveal top nsa hacking unit. http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html.

[4] The Slingshot Project. http://slingshot.mpi-sws.org.

[5] BAILEY, M., ANDERSEN, J., MORLEYMAO, Z., AND JAHANIAN, F. Automated classification and analysis of internet malware. In *Proceedings of Recent Advances in Intrusion Detection (RAID 2007)*.

[6] BILGE, L., AND DUMITRAS, T. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS 2012)* (2012).

[7] CITIZEN LAB. Surtr: Malware family targeting the tibetan community. Tech. rep. https://citizenlab.org/2013/08/surtr-malware-family-targeting-the-tibetan-community/.

[8] CONTAGIO. http://contagiodump.blogspot.de/2013/09/sandbox-miming-cve-2012-0158-in-mhtml.html.

[9] FIREEYE. Supply chain analysis: From quartermaster to sunshop. Tech. rep.

[10] HARDY, S., CRETE-NISHIHATA, M., KLEEMOLA, K., SENFT, A., SONNE, B., WISEMAN, G., AND GILL, P. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA).

[11] INFORMATION WARFARE MONITOR. Tracking ghostnet: Investigating a cyber espionage network. Tech. rep., 2009.

[12] JANA, S., AND SHMATIKOV, V. Abusing file processing in malware detectors for fun and profit. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy* (San Francisco, CA).

[13] KASPERSKY. The nettraveler (aka TravNeT). Tech. rep. https://www.securelist.com/en/downloads/vlpdfs/kaspersky-the-net-traveler-part1-final.pdf.

[14] KRUEGEL, C., KIRDA, E., COMPARETTI, P. M., BAYER, U., AND HLAUSCHEK, C. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)* (2009).

[15] MANDIANT. APT1 exposing one of chinas cyber espionage units. Tech. rep., 2013. http://intelreport.mandiant.com/.

[16] MARCZAK, W. R., SCOTT-RAILTON, J., MARQUIS-BOIRE, M., AND PAXSON, V. When governments hack opponents: A look at actors and technology. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA).

[17] NARAYANAN, A., PASKOV, H., GONG, N. Z., BETHENCOURT, J., CHUL, E., SHIN, R., AND SONG, D. On the feasibility of internet-scale author identification. In *Proceedings of the 33rd conference on IEEE Symposium on Security and Privacy. IEEE* (San Francisco, CA, 2012).

[18] NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov/.

[19] RALPH LANGNER. To kill a centrifuge: A technical analysis of what stuxnets creators tried to achieve. Tech. rep., 2013.

[20] REUTERS. Journalists, media under attack from hackers: Google researchers. www.reuters.com/article/2014/03/28/us-media-cybercrime-idUSBREA2R0EU20140328.

[21] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Hyderabad, India, 2008).

[22] SYMANTEC. 2013 internet security threat report, volume 18. Tech. rep. http://www.symantec.com/security_response/publications/threatreport.jsp.

[23] WIRED. Google hackers targeted source code of more than 30 companies. http://www.wired.com/2010/01/google-hack-attack/.

A Targeted organizations

Organization	*# Recipients*	*# Emails*	*First-Last*
World Uyghur Congress (WUC)	53	2,366	2009-2013
East Turkestan Union in Europe (ETUE)	7	153	2010-2013
Australian Uyghur Association	3	129	2009-2013
Euro-Asia Foundation in Turkey	2	101	2010-2013
Uyghur Canadian Association	6	98	2009-2013
Germany Uyghur Women Committee	2	82	2009-2013
Radio Free Asia (RFA)	12	80	2010-2013
France Uyghur Association	5	80	2009-2013
Eastern Turkestan Australian Association (ETAA)	6	77	2009-2013
Uyghur American Association (UAA)	10	72	2010-2013
Eastern Turkestan Uyghur Association in Netherlands	4	69	2010-2013
Netherland Uyghur Union	1	60	2012-2013
United Nations for a Free Tibet (UK)	1	57	2011-2013
Eastern Turkestan Culture and Solidarity Association	13	53	2009-2013
Viktoria Uyghur Association	1	48	2010-2013
Japan Uyghur Association	2	43	2012-2013
Switzerland East Turkestan Association	2	43	2010-2013
Hacettepe University Turkey	3	41	2010-2013
Kazakhstan Academy of Poetry	2	36	2009-2013
Belgium Uyghur Association	1	35	2009-2013
Kyrgyzstan Uyghur Association	2	33	2011-2013
Uyghur Canadian Society	1	31	2009-2013
Uyghur Academy	5	25	2009-2013
Munich Uyghur Elders Meshrep	1	22	2012-2013
Republican Uyghur Cultural Center of Kazakhstan	1	22	2009-2013
Sweden Uyghur Association	4	12	2010-2013
Virginia Department of Social Services	1	11	2009-2012
Unrepresented Nations and Peoples Organization (UNPO)	5	8	2010-2013
Sociale Verzekeringsbank (SVB) NGO Netherland	1	8	2012-2013
China Democratic Party (CDP)	5	5	2009-2011
Finland Uyghur Association	1	5	2013
Jet Propulsion Laboratory, founded by NASA	1	5	2012
Pennsylvania State University US	1	5	2010-2013
Uyghur Support Group Nederland	2	5	2010-2013
Norway Uyghur Committee	1	5	2010-2013
Amnesty International	4	4	2010-2012
Association of European Border Regions (AEBR)	1	4	2010-2011
Howard University US	1	4	2012-2013
Initiatives for China	3	4	2009-2010
LSE Asia Research Center and Silk Road Dialogue	2	4	2012-2013
The Government-in-Exile of the Republic of East Turkistan	2	4	2010-2011
Uyghur Human Rights Project (UHRP)	2	4	2010
Australian Migration Options Pty Ltd	3	4	2010
Agence France-Presse	1	3	2013
National Endowment for Democracy (NED)	2	3	2010-2012
PEN International	3	3	2009-2013
Syracuse University US	1	3	2013
Worldwide Protest in Honor and Support of Uyghurs Dying for Freedom	1	3	2013
Australian Government - Department of Foreign Affairs and Trade	2	3	2010
New Tang Dynasty Television China	2	3	2010
The Epoch Times	2	3	2010
Ministry of Foreign Affairs Norway	1	2	2013
International University of Kagoshima Japan	1	2	2013
Association of Islam Religion	1	2	2013
Bilkent University Turkey	2	2	2011-2012
Embassy of Azerbaijan in Beijing	2	2	2010
Indiana University School of Law-Indianapolis LL.M.	1	2	2012
KYOCERA Document Solutions Development America	1	2	2013
New York Times	2	2	2009
Pfizer Government Research Laboratory - Clinical Pharmacology	1	2	2011-2012
Saudi Arabia - Luggage Bags and Cases Company	1	2	2013
Students for a Free Tibet	2	2	2010
Sweden Uyghur Education Union	1	2	2010-2013
Uyghur International Culture Center	1	2	2012
The Protestant Church Amsterdam	1	2	2010
Swiss Agency for Development and Cooperation (SDC) Kargyzstan	1	1	2013
American Bar Association for Attorneys in US	1	1	2010
Assistance for Work Germany Frankfurt	1	1	2010
Bishkek Human Rights Committee	1	1	2012
Central Tibetan Administration (CTA)	1	1	2010
Chinese Translation Commercial Business	1	1	2009
Circassian Cultural Center (CHKTS)	1	1	2012
Colombian National Radio	1	1	2010
Embassy of the United States in Australia	1	1	2010
Europa Haber Newspaper Turkey	1	1	2010
Europe-China Cultural Communication (ECCC)	1	1	2011
Freelance Reporter and writer Turkey	1	1	2012
Goethe University Frankfurt am Main Germany	1	1	2012
Human Rights Campaign in China	1	1	2010
International Enterprise (IE) - Singapore Government	1	1	2010
International Tibet Independence Movement	1	1	2010
Jasmine Revolution China (Pro-Democracy Protests)	1	1	2009
Socialist Party (Netherlands)	1	1	2011
Los Angeles Times	1	1	2010
Milli Gazete (National Newspaper Turkey)	1	1	2010
Norwegian Tibet Committee	1	1	2010
Photographer Turkey	1	1	2012
CNN International Hong Kong	1	1	2012
Reporters Without Borders	1	1	2012
Republican National Lawyers Association Maryland	1	1	2010
Save Tibet - International Campaign for Tibet	1	1	2010
Society for Threatened People (STPI)	1	1	2012
Southern Mongolian Human Rights	1	1	2012
Stucco Manufacturers Association US	1	1	2013
Superior School of Arts France	1	1	2012
The George Washington University	1	1	2013
TurkishNews Newspaper	1	1	2010
US Bureau of Transportation Statistics	1	1	2009
Umit Uighur Language School	1	1	2010
Union of Turkish-Islamic Cultural Associations in Europe	1	1	2012
University of Adelaide Melbourne	1	1	2010
University of Khartoum Sudan	1	1	2012
US Embassy and Consulate in Munich Germany	1	1	2011
Wei Jingsheng Foundation	1	1	2009
Xinjiang Arts Institute China	1	1	2010
Yenicag Gazetti (Newspaper Turkey)	1	1	2010
American University	1	1	2012
Islamic Jihad Union	1	1	2012

B Dynamic taint-assisted analysis of malicious documents

B.1 Methodology

We use BitBlaze [21] to perform dynamic taint-tracking analysis of the targeted applications under the malicious documents as input and configure it to report four kinds of reports: (a) when a tainted Extended Instruction Pointer (EIP) is executed, (b) when a memory fault is triggered in the target program, (c) when a new process is spawned from the target program, or (d) when the analysis "times out" (i.e., runs without interruption for over 15 minutes). To mark malicious documents as a source of taint, we tainted the network inputs and routed the malicious input file using `netcat`. Additionally, we set BitBlaze to exit tracing at the detection of null pointer exceptions, user exceptions, tainted EIPs, and process exits before the start of the trace. A tag was generated from the trace by obtaining the last instruction with tainted operands, and matching it with the list of loaded modules generated by TEMU. Our guest (analysis) system configuration used in the image consists of clean installations of Windows XP SP2 with TEMU drivers and Microsoft Office 2003.

B.2 Results

Anti-virus software typically uses static signature-matching or whitelisting techniques to analyze malware. To validate the analysis results available from commercial AV, we ran a separate semi-automated dynamic analysis of the targeted application under our malicious documents.

Out of 817 unique input documents (725 malicious and 92 legitimate), 295 timed out with our BitBlaze analysis without reporting a tainted EIP, a memory fault, or a newly spawned process.[2] Another 13 of them were incompatible with our analysis infrastructure (using a more recent DOCX format). We could not compare these cases directly to the results obtained from VirusTotal. Therefore, we focus on the remaining 509 malicious documents in the evaluation.

Efficacy of Taint EIP Detection. Taint-tracking detected tainted EIP execution in 477 out of the 509 documents. In 19 cases of the undetected 32 cases, however, a new process was spawned without it being detected by taint-tracking. We treat these as false negatives in taint-tracking. We speculate that this is likely to be due to missed direct flows, untracked indirect flows (via control dependencies, or table-lookups), or attacks using non-control-flow hijacking attacks (such as argument corruption). 13 documents did not lead to a tainted EIP execution, but instead caused a memory fault. This could be due to a difference in our test infrastructure and the victim's, or an attempt to evade analysis. In 33 of the 477 cases where tainted EIP was detected, no new spawned process was created, and the tainted EIP instruction did not correspond to any shellcode. All these cases correspond to a particular instruction triggering the tainted EIP detection in MSO.DLL, a dynamic-link library found in Microsoft Office installations. To understand this case better, we manually created blank benign documents and fed them to Microsoft Office — they too triggered tainted EIP detections. We treat these cases as false positives in taint detection, possibly because of benign dynamic generation of code. All the remaining cases (i.e., 444 out of 477) are legitimate exploits that we could confirm to execute shellcode.

[2] We believe 150 of these are due to user-interaction which we could not presently automate, and the remaining could potentially be analyzed with a faster test platform; we plan to investigate this in the future.

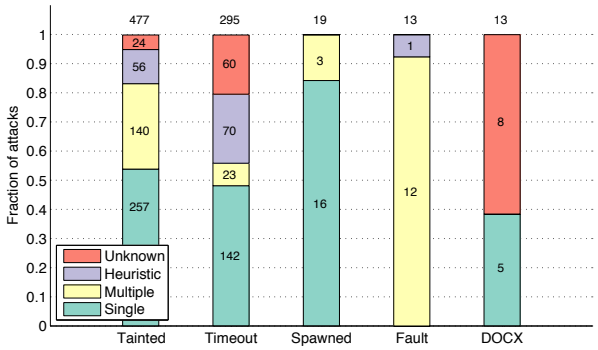


Figure 12: Breakdown of dynamic taint-assisted analysis, and comparison to VirusTotal AV results. *Single*, *Multiple*, *Heuristics*, and *Unknown* correspond to the different AV tags assigned to documents. The main bars show the detection result from BitBlaze: (a) Detected by *Tainted* EIP execution, (b) *Timeout*, (c) *Spawned* process without tainted EIP execution, (d) Memory *Fault* without tainted EIP execution, and (e) *DOCX* unable to run in our analysis environment. Within each main bar, each stacked bar represents the corresponding tag given by VirusTotal.

Dynamic Taint versus VirusTotal. Figure 12 shows the detailed comparison of taint-assisted classification of vulnerabilities versus the results from VirusTotal. Out of a total of 477 documents on which tainted EIP was detected, VirusTotal tagged 397 documents with one or more CVEs. Of the remaining 80 cases that are detected by tainted EIP execution, 24 are undetected by VirusTotal, and 56 are detected, but marked *Unknown* (i.e., no CVE assigned) by VirusTotal. Dynamic taint analysis to determine the tainted EIP was helpful to further refine the results of AV detection for a majority of these 56 tagged-*Unknown* cases. Specifically, for 55 out of the 56 documents, taint-assisted manual analysis was able to resolve it to the exploited CVE.

Out of a total of 477 documents on which tainted EIP was detected, VirusTotal tagged 397 documents with one or more CVEs. Our taint-assisted manual analysis agrees with the VirusTotal CVE tag results on 372 of these 397. That is, 372 documents were detected to execute a tainted EIP for which we could manually correlate to a single CVE that was the same as the one reported by a majority of the AVs in VirusTotal.[3] Thus, for a large majority of the cases, taint-assisted analysis agrees with the AV results. Of the remaining 25 cases, 17 could be identified as misclassifications because the CVE reported by most of the AVs in VirusTotal was not the one that affected the program. The 8 remaining documents were tagged by taint analysis as being false positives even though a CVE was obtained from VirusTotal.

[3] Note that different AVs often tag the same vulnerability with different tags in VirusTotal. We took the tag given by a majority of the reported tags, as the representative of the sample.

C Command and Control (C2) servers

C2	# Emails	C2	# Emails	C2	# Emails	C2	# Emails
61.178.77.169	74	mzyzy.vicp.net	3	www.info-microsoft.com	2	googlehk.dynamicdns.co.uk	1
dtl.dnsd.me	66	mygoodbug.dnsd.info	3	www.uyhanur.nna.cc	2	113.10.201.254	1
ns.dns3-domain.com	55	www.uyghuri.mrface.com	3	www.micosofts.com	2	152.101.38.177	1
dtl.eatuo.com	44	6.test.3322.org.cn	3	100.4.43.2	2	blog.sina.com.cn	1
202.85.136.181	32	218.82.206.229	3	61.234.4.214	1	uyghur.epac.to	1
update.googmail.org	31	uyghur.sov.tw	3	a.yahoohello.com	1	xinxin20080628.gicp.net	1
dtl6.mooo.com	29	3.test.3322.org	3	bc1516.7766.org	1	yah00mail.gicp.net	1
www.discoverypeace.org	26	newwhitehouse.org	3	202.68.226.250	1	hbnjx.6600.org	1
58.64.172.177	22	goodnewspaper.f3322.org	3	msdn.homelinux.org	1	humanbeing2009.gicp.net	1
email.googmail.org	22	nskupdate.com	3	207.204.245.192	1	webhelp01.freetcp.com	1
news.googmail.org	22	webmonder.gicp.net	3	216.131.66.96	1	mobile.yourtrap.com	1
61.128.122.147	17	61.132.74.68	3	www.avasters.com	1	125.141.149.23	1
softmy.jkub.com	15	61.178.77.108	3	202.130.112.231	1	222.73.27.223	1
61.234.4.213	13	betterpeony.com	3	nbsstt.3322.org	1	www.jiapin.org	1
dnsmm.bpa.nu	11	4.test.3322.org	3	goodnewspaper.3322.org	1	ibmcorp.slyip.com	1
121.170.178.221	10	61.234.4.210	3	webposter.gicp.net	1	182.16.11.187	1
zeropan007.3322.org	10	9.test.3322.org.cn	3	uyghur1.webhop.net	1	star2.ksksz.com	1
wwzzsh.3322.org	9	8.test.3322.org.cn	3	webwx.3322.orgxiexie.8866.org	1	69.197.132.130	1
222.77.70.237	9	1.test.3322.org	3	125.141.149.49	1	www.yahooprotect.com	1
3.test.3322.org.cn	8	radio.googmail.org	3	guanshan.3322.org	1	xiexie.8866.org	1
1.test.3322.org.cn	8	7.test.3322.org.cn	3	leelee.dnset.com	1	img.mic-road.com	1
2.test.3322.org.cn	8	tokyo.collegememory.com	2	uygur.eicp.net	1	photo.googmail.org	1
eemete.freetcp.com	8	201.22.184.42	2	kxwss.8800.org	1	tonylee38.gicp.net	1
apple12.crabdance.com	8	61.178.77.96	2	173.208.157.186	1	suggest.dns1.us	1
wolf001.us109.eoidc.net	7	webproxy.serveuser.com	2	rc.arkinixik.com	1	worldview.instanthq.com	1
4.test.3322.org.cn	7	www.bbcnewes.net	2	www.uusuanru.nna.cc	1	goodnewspaper.gicp.net	1
etdt.cable.nu	6	done.youtubesitegroup.com	2	uxz.fo.mooo.com	1	112.121.182.150	1
205.209.159.162	6	alma.apple.cloudns.org	2	uygur.51vip.biz	1	abc69696969.vicp.net	1
br.stat-dns.com	6	webmailsvr.com	2	peopleunion.gicp.net	1	put.adultdns.net	1
66.79.188.23	6	polat.googmail.org	2	free1000.gnway.net	1	loadbook.strangled.net	1
www.southstock.net	6	religion.xicp.net	2	uxz.fo.dnsd.info	1	internet.3-a.net	1
ns1.3322.net	5	connectsexy.dns-dns.com	2	wodebeizi119.jkub.com	1	news.scvhosts.com	1
121.254.173.57	5	dns3.westcowboy.com	2	itsec.eicp.net	1	98.126.20.221	1
www.uyghur.25u.com	5	61.220.138.100	2	stormgo.oicp.net	1	mydeyuming.cable.nu	1
202.96.128.166	5	27.254.41.7	2	boy303.2288.org	1	gshjl.3322.org	1
ns1.oray.net	5	116.92.6.197	2	webjz.9966.org	1	forever001.dtdns.net	1
jhska.cable.nu	5	apple12.co.cc	2	zbing.strangled.net	1	grt1.25u.com	1
test195.3322.org	5	58.64.129.149	2	tommark5454.xxxy.info	1	66.197.202.242	1
61.234.4.218	5	worldmaprsh.com	2	oyghur1.webhop.net	1	kaba.wikaba.com	1
61.128.110.37	5	phinex127.gicp.net	2	addi.apple.cloudns.org	1	221.239.96.180	1
ns1.china.com	5	wxjz.6600.org	2	60.170.255.85	1	174.139.133.58	1
a2010226.gicp.net	5	gecko.jkub.com	2	toolsbar.dns0755.net	1	125.141.149.46	1
logonin.uyghuri.com	4	smtp.126.com	2	61.132.74.113	1	frank.3feet.com	1
macaonews.8800.org	4	errorslog.com	2	113.10.201.250	1	115.126.3.214	1
book.websurprisemail.com	4	uyghurie.51vip.biz	2	home.graffiti.net	1	liveservices.dyndns.tv	1
desk.websurprisemail.com	4	tanmii.gicp.net	2	statistics.netrobots.org	1	inc.3feet.com	1
test.3322.org.cn	4	211.115.207.7	2	freesky365.gnway.net	1	1nsmm.bpa.nu	1
221.239.82.21	4	59.188.5.19	2	greta.ikwb.com	1	www.yahooprotect.net	1
liveservices.dyndns.info	4	206.196.106.85	2	englishclub.2288.org	1	222.82.220.118	1
180.169.28.58	4	religion.8866.org	2	mm.utf888.com	1	webwxjz.3322.org	1
portright.org	4	68.89.135.192	2	annchan.mrface.com	1	61.234.4.220	1
video.googmail.org	4	blogging.blogsite.org	2	www.shine.4pu.com	1	thankyou09.gicp.net	1
www.guzhijiaozihaha.net	4	softjohn.ddns.us	2	copy.apple.cloudns.org	1	218.28.72.138	1
207.46.11.22	4	report.dns-dns.com	2	220.171.107.138	1	soft.epac.to	1
www.googmail.org	4	115.160.188.245	2	uyghuri.mrface.com	1	www.yahooip.net	1
2.test.3322.org	3	newyorkonlin.com	2	218.108.42.59	1	msejake.7766.org	1
dcp.googmail.org	3	tw252.gicp.net	2	58.64.193.228	1	202.67.215.143	1
test.3322.org	3	61.222.31.54	2	tt9c.2288.org	1	www.yahoohello.com	1
np6.dnsrd.com	3	tomsonmartin.ikwb.com	2	forum.universityexp.com	1	202.109.121.138	1

A Large-Scale Empirical Analysis of Chinese Web Passwords

Zhigong Li, Weili Han
Software School, Fudan University
Shanghai Key Laboratory of Data Science, Fudan University

Wenyuan Xu
Department of Electronic Engineering, Zhejiang University

Abstract

Users speaking different languages may prefer different patterns in creating their passwords, and thus knowledge on English passwords cannot help to guess passwords from other languages well. Research has already shown Chinese passwords are one of the most difficult ones to guess. We believe that the conclusion is biased because, to the best of our knowledge, little empirical study has examined regional differences of passwords on a large scale, especially on Chinese passwords. In this paper, we study the differences between passwords from Chinese and English speaking users, leveraging over 100 million leaked and publicly available passwords from Chinese and international websites in recent years. We found that Chinese prefer digits when composing their passwords while English users prefer letters, especially lowercase letters. However, their strength against password guessing is similar. Second, we observe that both users prefer to use the patterns that they are familiar with, *e.g.*, Chinese Pinyins for Chinese and English words for English users. Third, we observe that both Chinese and English users prefer their conventional format when they use dates to construct passwords. Based on these observations, we improve a PCFG (Probabilistic Context-Free Grammar) based password guessing method by inserting Pinyins (about 2.3% more entries) into the attack dictionary and insert our observed composition rules into the guessing rule set. As a result, our experiments show that the efficiency of password guessing increases by 34%.

1 Introduction

Passwords are the most widely used credentials for authenticating Web users around the world, including the users that do not speak English. Text-based passwords are likely to remain the dominant mechanism for authenticating users for the foreseeable future [7][19]. Meanwhile, researchers are still in the process of understanding the security strength of passwords and exploiting methods to improve password guessing. Although insightful, most existing work focuses on passwords of English users. Little work has studied the impact of regional convention and languages on password selection utilizing a large dataset of passwords. One exception is Bonneau [6], who studied password strength based on languages by performing an empirical study on Yahoo! users and concluded that Chinese passwords are among the hardest ones to guess. We believe his finding is biased because of his dataset (*i.e.*, `Yahoo` users are familiar with English). In this paper, we analyze passwords of non-English speakers, specifically, Chinese users, which represent *618* million Internet users as of the end of 2013 [12], and compare them with passwords of English users.

To understand the differences between Chinese and English passwords, this paper leverages over 100 million leaked and publicly available passwords from several popular Chinese websites (`CSDN` [13], `Tianya` [33], `Duduniu` [17], `7k7k` [5], and `178.com` [4]) and English websites (`RockYou` [30] and `yahoo` [37]). These Chinese websites only provide Chinese webpages, and we consider their users as *Chinese users*. In addition, English websites mainly intend to serve users who are familiar with English, and we consider the users of `RockYou` and `Yahoo` as *English users*. Note that, these websites (except `Duduniu`, which is an e-commerce website) provide similar services, i.e., non-monetary ones such as web portal, online communities, social networking, online forums, etc. Thus, we consider them comparable and having similar influence on their users when choosing passwords. This makes their password data corpus promising for studying the impact of languages on password composition.

The unfortunate leakages of the large volume of passwords provides us an opportunity to understand password differences between the two groups of users in depth. Such analysis is important, because it enables

	Language	Site Address	Amount	Distinct Accounts
CSDN	Chinese	http://www.csdn.net/	6,428,629	6,423,483
Tianya	Chinese	http://www.tianya.cn/	30,179,474	26,223,020
Duduniu	Chinese	http://www.duduniu.cn/	16,282,969	15,131,833
7k7k	Chinese	http://www.7k7k.com/	19,138,270	15,940,099
178.com	Chinese	http://www.178.com/	9,072,824	9,072,804
RockYou	English	http://www.rockyou.com/	32,603,048	32,602,882
Yahoo	English	http://www.yahoo.com/	442,837	442,837
Total			**114,148,051**	**105,836,958**

Table 1: Basic information of leaked passwords of the websites that are analyzed in this paper. We removed the duplicated accounts between `Tianya` and `7k7k` from the `Tianya` dataset. See details in Appendix A.

better password guessing evaluation and can guide web masters to protect the accounts.

We designed analysis tools and leveraged the guessing resistance indicators (such as α-*work-factors* [28] and β-*success-rates* [10]) to find the differences among accounts of multiple websites, and found the preference of the two groups of users. Then, we improved the efficiency of the Probabilistic Context-Free Grammar (PCFG) based password guessing method [35] by adding regionally preferred patterns (*i.e.* Pinyins) into the dictionary and modifying the generated guessing rules. We summarize our findings and main contributions as follows:

- **Different Characters Sets**: Chinese users prefer digits in their passwords, while English users prefer letters, especially lowercase letters. However, the password strength against guessing is similar for both groups and thus both groups share similar security concerns in protecting passwords.
- **Patterns of Languages and Dates**: Both Chinese and English users prefer to use language-related patterns as passwords. That is, Chinese users prefer Chinese Pinyins and English users prefer English words. As for dates, both groups prefer their conventional formats. That is, Chinese prefer dates with the year at the beginning and English users prefer dates with the year at the end.
- **Improvement of the Efficiency of Password Guessing**: Based on our observations, we add 20,000 Pinyins into the dictionary and add the guessing rules, resulting in an improvement of efficiency by 34% in guessing Chinese passwords using a PCFG based guessing method. This confirms that the Pinyins and date's rules are important in guessing Chinese passwords.

The rest of the paper is organized as follows: Section 2 summarizes our observations on the differences between passwords from Chinese and English users. Section 3 presents the results of guessing using modified Bonneau's methods [6] and PCFG based methods [35]. In Section 4, we discuss the related work and conclude in Section 5.

2 Regional Differences on Passwords

2.1 Dataset Setup

To discover the differences between the passwords of Chinese and English users, we analyzed a corpus of over 100 million passwords from multiple websites that are in Chinese and English, respectively. All the leaked passwords are publicly available for downloading. During our research, we followed the ethical practice and never utilized the leaked passwords for reasons other than understanding the overall statistical observation of passwords.

At the end of 2010, an incident that is known as *CSDN Password Leakage Incident* happened, and passwords from five websites, including `CSDN`, `Tianya`, `Duduniu`, `7k7k` and `178.com`, were leaked in several consecutive days. The total number of leaked accounts is over 80 million, and all the leaked passwords are in plaintext. We summarize the website information in Table 1.

`CSDN` [13] is one of the most popular Chinese IT professional communities, similar to `MSDN`. `Tianya` [33] is the largest online forums and blogs in China. `7k7k` [5] and `178.com` [4] are two websites providing game infor-

	Chinese	English
1	123456 (2.17%)	123456 (0.88%)
2	123456789 (0.65%)	12345 (0.24%)
3	111111 (0.59%)	123456789 (0.23%)
4	12345678 (0.39%)	password (0.18%)
5	000000 (0.34%)	iloveyou (0.15%)

Table 2: The most popular passwords and their occurrence percentages.

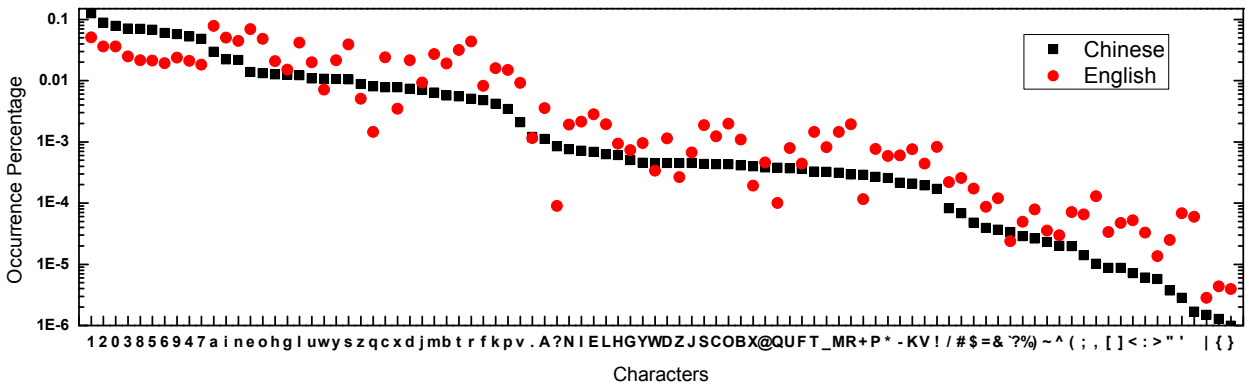


Figure 1: Character distribution, i.e., the occurrence percentage of each character for Chinese and English passwords. The characters are arranged in a descending order according to the percentages in Chinese passwords.

mation and online flash games. Duduniu [17] is a commercial site that mainly sells management software platforms for Internet bars. It is worth noting that all these websites are extremely popular in China, among which CSDN and Tianya have been ranked top 1,000 in Alexa Top Global Sites recently. Thus, their users cover a large percentage of Internet users in China.

Besides their popularity, the leaked password data corpus is promising for understanding the language impact on passwords because few password policies are enforced in the above five Chinese websites before the leakage according to our investigation. For example, CSDN allows a password with as few as five digits, and such a rule remains unchanged even after the password leakage event. Furthermore, Tianya allows passwords as short as six characters since it was founded. Thus, the leaked password data corpus represents the password set that was composed with little influence from password policies.

Password leakage events also happened to English websites as well. In 2009, attackers broke into the database of RockYou and released the 32 million passwords (in plaintext) to the public. In 2012, Yahoo's accounts were leaked. A hacking group 'DD3Ds Company' utilized a union-based SQL injection to obtain login details of about 450 thousand user accounts.

The raw files contain duplication and blank passwords that can affect the analysis. For instance, we detected that attackers copied a portion of 7k7k passwords to Tianya, because the password duplication rate between Tianya and 7k7k is much more than the rate between any other two websites (i.e., about 90% between Tianya and 7k7k and about 30% between any other two websites). We thus removed these duplicate passwords in Tianya using the method described in Appendix A. After removing the accounts with blank passwords and filtering out duplicated accounts, we obtained 105,836,958 accounts, as detailed in Table 1. Finally, we imported them into MySQL for further analysis.

2.2 Password Comparison

2.2.1 The Most Popular Passwords

We list the five most popular passwords of Chinese and English users in Table 2, from which we have the following observations:

- In total, the five most popular passwords constitute 4.14% of all Chinese passwords and 1.69% of all English passwords, which shows that Chinese passwords are more congregated.
- Interestingly, although in English datasets, there are a larger number of letter-only passwords (see details in Section 2.2.3), the top 3 most popular passwords are digit-only. In addition, both groups share similar popular passwords, *e.g.*, 123456 and 123456789.

2.2.2 Character Distribution

To understand the frequency of each character, which includes letters (a-z, A-Z), digits (0-9), and symbols (all printable characters except digits and letters), we analyzed the percentage of each character for Chinese and English passwords and depict them in Figure 1, where the characters are arranged in descending order according to the percentages in Chinese passwords.

- **Digits**. In Chinese passwords, the top used characters are digits. Although English users do not use digits as frequently as Chinese users do, digits are among the most frequently used characters.
- **Letters**. In general, Chinese passwords use letters less frequently than English passwords do. In addition, some letters exhibit similar usage percentages

	Digit -only	Letter-only (Lowercase-only)	Letter+Digit (Lowercase+Digit)	Letter+Symbol (Lowercase+Symbol)	Symbol +Digit	Letter+Digit+Symbol (Lowercase+Digit+Symbol)
CSDN	45.06%	12.39% (11.68%)	39.02% (35.60%)	0.50% (0.42%)	0.61%	2.39% (2.04%)
Tianya	64.56%	10.20% (9.89%)	23.12% (21.27%)	0.25% (0.22%)	0.71%	1.14% (1.01%)
Duduniu	32.86%	11.76% (11.08%)	53.69% (50.93%)	0.52% (0.48%)	0.17%	0.92% (0.80%)
7k7k	60.77%	11.13% (10.75%)	26.41% (23.03%)	0.14% (0.12%)	0.32%	1.14% (0.49%)
178.com	48.07%	9.17% (9.00%)	42.11% (41.25%)	0.06% (0.06%)	0.31%	0.27% (0.26%)
RockYou	15.93%	44.04% (41.68%)	36.22% (33.17%)	1.91% (1.64%)	0.16%	1.71% (1.44%)
Yahoo	5.89%	34.64% (33.08%)	56.62% (50.60%)	0.62% (0.49%)	0.04%	2.18% (1.38%)

Table 3: Compositions of passwords. The percentages outside parentheses are the ones counting both uppercase and lowercase letters, and the percentage inside parentheses are the ones counting only lowercase letters. The sum of the percentages in one row is slightly smaller than one, because symbol-only passwords are not listed, and they only account for a small percentage.

	# of Structures/10K	Most Popular Structure	Most Popular Structure%
CSDN	884	DDDDDDDD	21.50%
Tianya	756	DDDDDD	30.10%
Duduniu	610	DDDDDD	7.25%
7k7k	635	DDDDDD	19.51%
178.com	459	DDDDDD	15.48%
RockYou	803	LLLLLL	5.40%
Yahoo	1165	LLLLLL	9.19%

Table 4: Structures of passwords. # of structures/10K refers to the number of different structures in every 10,000 passwords, and the other two columns contain the structures and occurrence percentages of the most popular passwords in both Chinese websites and English ones. D represents a digit, and L represents a lowercase letter.

for both groups of passwords, *e.g.*, the letter *a* is the mostly used letter in both groups. Some letters show distinct usages, *e.g.*, the letter *q* is frequently used in Chinese passwords but is much less used in English passwords; the letter *r* is much more popular in English passwords than in Chinese ones. This is because of the word patterns in either languages. For instance, the letters *q* and *a* are popular building blocks of Pinyins, but the letter *r* is not. We will discuss Chinese Pinyins and English words in detail in Section 2.2.5.

- **Symbols**. Symbols are used less in both Chinese and English passwords, in general. Interestingly, for both groups of passwords, several symbols share the similar usage percentages: the symbol dot (.) is the most frequently used, and symbols like left brace ({) and right brace (}) are less likely to be used. However, regional differences on symbol usages do exist: the question mark (*?*) is more frequently used in Chinese passwords than in English passwords.

2.2.3 Compositions and Structures of Passwords

To understand the structures of passwords in both groups, we analyzed passwords in two aspects. (1) we divided passwords according to their compositions and calculated the percentages in seven category (shown in Table 3). The categories are pure digits, pure letters, digits and letters, letters and symbols, etc. (2) We calculated the percentages of different types of password structures utilizing representations in the Probabilistic Context-Free Grammar [35]. For example, the structure of *Johns0n!* is modeled as *ULLLLDLS* (U = uppercase, L= lowercase, D = digit, and S = symbol). The structure comparison of both password groups is shown in Table 4 where *# of Structures/10K* refers to the number of different structures in every 10,000 passwords. *The most popular structure* is the one that appears the most in the data-set. From Table 3 and Table 4, we can obtain the following observations:

- A majority (around 50% on average) of Chinese users prefer digit-only passwords. This could be due to their language. Chinese characters cannot be entered directly as a password, and digits appear to be the best candidate when users are creating new passwords. Although Chinese users can use Pinyins as discussed in Section 2.2.5, digits seem to be more convenient. As shown in Table 4, *DDDDDD* is the dominant structure in most Chinese websites. For

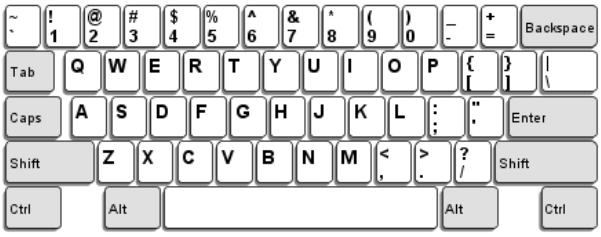


Figure 2: A typical layout of a keyboard (103P) used in China, which is the same as an English keyboard layout.

CSDN, the structure *DDDDDDDD* is the top selection, and *DDDDDD* is ranked at 14. A six-digit number may be an ATM PIN, a birthday, or the last six digits of citizen ID cards. We will discuss details in Section 2.2.6.

- For both password groups, a good portion of passwords contain both letters and digits, and no obvious differences seem to exist between these websites. The owners of the passwords in this category could be users who are concerned with password security but are unwilling to bother with symbols.

2.2.4 Keyboard Patterns

Sometimes, users prefer to create their passwords according to keyboard patterns [32]. Thus, we analyzed the percentages of three primary keyboard patterns. Note that Chinese users utilize standard English keyboards (shown in Figure 2), i.e., they use the same ones as English users.

- **Same Row**: The same row passwords are formed by a consecutive sequence of characters in the same row on keyboard, *e.g.*, asdfhj.
- **Zig Zag**: The zig-zag passwords are formed by a sequence of characters, where each key is adjacent to the next one but not in the same row, *e.g.*, qawsxd.
- **Snake**: The snake passwords consist of a sequence of characters whose keys are adjacent on keyboards yet they are neither in the *Same Row* or *Zig Zag*, *e.g.*, zxcfgh.

	Chinese	English
Same Row	8.31% (0.55%)	2.42% (0.25%)
Zig Zag	0.26%	0.06%
Snake	0.27%	0.08%

Table 5: Percentage of passwords with different keyboard patterns. Most passwords of the *Same Row* pattern are digit-only. The numbers in the parentheses represent passwords that have the *Same Row* pattern and are not digit-only.

Algorithm to Identify Keyboard Patterns. In order to automatically classify passwords into the aforementioned three categories, we assign a coordinate to each character on the keyboard. We define that the x-axis increases from left to right and the y-axis increases from top to bottom. For example, the coordinates of 1 (and !) are (1,0), and the coordinates of q, a, and z are (1,1), (1,2), and (1,3), respectively. Provided the coordinates of the characters, we can determine if a password is in a specific keyboard pattern using the algorithm illustrated in Algorithm 1, where $isAdjacent(pos1, pos2)$ determines whether two letters located in the coordinates $pos1$ and $pos2$ are adjacent in the same row or column.

Result. The statistics analyzed by Algorithm 1 is shown in Table 5, from which we observe that more than 8% of Chinese passwords are composed according to keyboard patterns but fewer English passwords are. After removing all digit-only passwords, the keyboard pattern passwords reduce to about 1%. This is because most passwords of the *same row* pattern are digit only. Nevertheless, Chinese users tend to use keyboard pattern passwords more often than English users do, e.g., there are 0.2% more *Zig Zag* passwords for Chinese than English users. This could be because keyboard patterns are easy to create and remember for Chinese users who are unfamiliar with English.

2.2.5 Chinese Pinyins and English Words

Chinese Pinyin was developed in 1950s and is designed to represent the pronunciation of Chinese characters. Although there are lots of dialects in China, the Pinyins for characters are the same. Trained with Pinyin since primary school, Chinese computer users are familiar with it. Pinyin is the most popular method to input Chinese characters to a computer because it requires almost no extra training for Chinese. Typically, a Chinese character is entered by multiple keystrokes. Although other input methods, such as Wubi, exist, these methods are not as popular due to their steep learning curves.

Since websites do not support passwords composed of Chinese characters directly, unsurprisingly, just like the words in English passwords, Pinyins are widely used in passwords of Chinese users. Ignoring the tones, typically, a word in Pinyins uses a set of 21 sounds representing the beginning of the word called initials, and a set of 37 sounds representing the end of the word called finals. These two combine to form about 420 different basic Pinyin elements [3]. However, users may use various compositions of multiple Pinyins in their passwords. For example, the password *nihao*, is composed of Pinyins *ni* and *hao*.

	Letter-only Passwords		Mixed Passwords	
	Chinese Pinyins%	English Words%	Chinese Pinyins%	English Words%
CSDN	41.61% (5.15%)	15.59% (1.93%)	25.49% (10.68%)	7.97% (3.34%)
Tianya	40.63% (4.15%)	10.39% (1.06%)	23.59% (5.78%)	6.05% (1.48%)
Duduniu	33.28% (3.91%)	15.35% (1.80%)	25.17% (13.87%)	6.48% (3.57%)
7k7k	44.70% (4.97%)	10.04% (1.12%)	21.09% (5.84%)	7.02% (1.94%)
178.com	57.31% (5.25%)	2.20% (0.20%)	23.49% (9.97%)	4.58% (1.94%)
RockYou	6.94% (2.99%)	25.47% (10.98%)	6.88% (2.61%)	28.11% (10.65%)
Yahoo	4.31% (1.46%)	34.92% (11.86%)	4.53% (2.59%)	27.99% (16.01%)

Table 6: Percentage of the passwords that contain Chinese Pinyins or English words. Mixed passwords refer to the ones that contain at least two types of characters with one of them being letters. The percentages inside the parentheses are the proportions out of the entire password dataset, and the percentage ahead of the parentheses are the ones out of the letter-only passwords or mixed passwords. For example, in the row of CSDN, 41.61% (5.15%) means that in the letter-only passwords, 41.61% are composed of Chinese Pinyins, and these passwords occupy 5.15% in the whole dataset of CSDN.

	Top Chinese Pinyins	Top English Words
1	woaini (1.47%)	password (1.28%)
2	li (1.06%)	iloveyou (0.98%)
3	wang (0.97%)	love (0.76%)
4	tianya (0.89%)	angel (0.59%)
5	zhang (0.84%)	monkey (0.45%)

Table 7: The most popular Chinese Pinyins and English words. The percentage base for top Chinese Pinyins is all the Pinyins we extracted from letter-only and mixed passwords in five Chinese websites. Similarly, the percentage base for top English words is all the words we extracted from letter-only and mixed passwords in both English websites.

Algorithm to Identify Pinyins or English Words. We can determine whether a password is composed of Chinese Pinyins or English words by string matching. For example, a password *helloworld* is composed of English words *hello* and *world*. For English words, we chose the Oxford English Dictionary [1] and extracted more than 20,000 commonly used English words.

To improve the matching efficiency, we use Trie (or prefix tree) to identify if the passwords are composed of Chinese Pinyins or English words. We first construct Trie by inserting Chinese Pinyins or English words one by one. With the Tries, we can identify if a password is composed of Chinese Pinyins or English words. The algorithm to insert entries into the Trie is shown in Algorithm 2. In our experiments, we constructed two Tries: one is constructed out of Chinese Pinyins, and the other is built based on the more than 20,000 commonly used English words. The procedure to identify if a password is composed of Chinese Pinyins or English words is shown in Algorithm 3. The structure *node* has two properties. The first is named as *child*, which is an array of *node* and represents the child nodes. The second is a boolean, *isValue*, which represents if the string from the root to the current node is a valid value. The algorithm will try to match the password with the known strings from Trie recursively.

Note that because it is hard to determine the semantic meaning, a password may be semantically meaningless even if it is a composition of Chinese Pinyins or English words. Furthermore, some passwords can be interpreted as compositions of Pinyins and English words at the same time. We removed the passwords with both Pinyins and English words in our analysis.

Result. We performed statistical analysis of the usage of Chinese Pinyins and English words in two aspects. Firstly, we calculated the percentages of passwords that are composed of Chinese Pinyins or English words out of all the letter-only passwords. Secondly, we calculated the percentages of Pinyins or English words out of all the mixed passwords (*i.e.*, the ones contain at least two types of characters with one of them being letters). The results are shown in Table 6. Table 7 lists the top five most popular Chinese Pinyins and English words. From Table 6 and 7, we draw the following conclusions:

- Out of the letter-only passwords, Pinyins are the dominant patterns for Chinese users in composing their passwords, and English words dominate the English passwords. Even when we consider all categories of passwords, these patterns are still the basic building blocks for a large portion of passwords, *i.e.*, more than 10% English passwords contain English words, and about 5% of Chinese passwords consist of Pinyins.

- Interestingly, it seems that love is always the main theme of human beings. As shown in Table 7, *love*

	# Consecutive Exactly Eight Digits	YYYYMMDD	MMDDYYYY	DDMMYYYY
CSDN	1,621,954	29.24%	0.25%	0.43%
Tianya	3,639,517	36.26%	0.35%	0.60%
Duduniu	1,700,329	28.87%	0.28%	0.84%
7k7k	2,470,204	32.41%	0.18%	0.37%
178.com	995,832	30.46%	0.13%	0.19%
RockYou	929,987	2.64%	7.70%	17.66%
Yahoo	6,981	2.78%	12.00%	11.17%

Table 8: Statistics of **eight-digit** date patterns: the number of occurrences of eight consecutive digits and percentages of three date formats. The percentage bases are listed in the second column. Y=year, M=month and D=day. For example, 20130115 is in the format of *YYYYMMDD*.

	# Consecutive Exactly Six Digits	YYMMDD	MMDDYY	DDMMYY
CSDN	809,050	27.21%	4.04%	1.24%
Tianya	9,477,069	23.93%	3.05%	1.19%
Duduniu	2,688,347	17.84%	2.97%	1.78%
7k7k	3,999,958	24.34%	2.63%	0.88%
178.com	2,525,254	13.96%	1.72%	1.30%
RockYou	2,758,871	5.63%	21.90%	18.42%
Yahoo	21,020	4.66%	25.99%	7.77%

Table 9: Statistics of **six-digit** date patterns: the number of occurrences of six consecutive digits and percentages of three date formats. The percentage bases are listed in the second column.

and *iloveyou* are ranked at the second and the third in English passwords. Meanwhile, *woaini* is the top ranked Pinyin, which means *I love you* in Chinese.

- The Pinyins of names are widely used in Chinese passwords. The Pinyins *li*, *wang* and *zhang*, listed as the top used Pinyins for passwords in Table 7, are among the most popular surnames in China. Note that it is difficult to identify first names in Chinese, because they could be almost any combinations of Pinyins.

- The website names appear to be an important part of Chinese passwords. For example, *tianya*, which is the website name, is ranked at the fourth in Chinese Pinyins.

- We found that some passwords from `RockYou` and `Yahoo` are composed of Pinyins, and we suspect that the owners are Chinese. Most of these Pinyins do not map to meaningful expression, and thus we suspect they are names. For example, *yaowei*, which is composed of Pinyins *yao* and *wei*, is most likely to be a name because either *yao* or *wei* can be a surname.

The influence of Chinese Pinyins in password guessing is discussed in Section 3.2.

2.2.6 Dates

Given that digits are commonly used in passwords, we try to understand the meaning of these digits. Since dates are typically represented as a string of digits, in this subsection we analyze the usage of dates in passwords.

Date Format. We focused our attention on six-digit and eight-digit dates. We first extracted all consecutive sequences of exactly six or eight digits from these passwords, and then calculated the dates which are in the range from 1900 to 2099. We classified six-digit dates into three formats: *YYMMDD*, *MMDDYY*, and *DDMMYY*. Similarly, we classify eight-digit dates into *YYYYMMDD*, *MMDDYYYY* and *DDMMYYYY*. The results are shown in Table 8 and 9. Note that there might be ambiguity when interpreting dates. For example, *11121987* may be interpreted as either *November 12, 1987* or *December 11, 1987*. In this case, we assigned the passwords to one of the formats according to the probability distribution of all the passwords that can be uniquely determined. For instance, if 20% of passwords that contain date can be uniquely identified as *MMDDYY* and 80% of them as *DDMMYY*. Then, we assigned 20% of the ambiguous passwords to *MMDDYY* and 80% to *DDMMYY*.

Furthermore, there may be false positive where a general six-digit number is considered as a date. For example, *123123* could be considered as *December 31, 1923*,

	Digit-only	Letter+Digit (Lowercase+Digit)	Symbol+Digit	Letter+Digit+Symbol (Lowercase+Digit+Symbol)
CSDN	51.98%	45.59% (41.36%)	0.50%	1.93% (1.67%)
Tianya	78.84%	19.91% (18.69%)	0.31%	0.72% (0.65%)
Duduniu	41.28%	58.17% (54.86%)	0.24%	0.31% (0.30%)
7k7k	73.90%	25.51% (24.61%)	0.18%	0.41% (0.37%)
178.com	50.91%	48.73% (48.07%)	0.32%	0.04% (0.04%)
RockYou	82.62%	16.52% (14.99%)	0.23%	0.63% (0.54%)
Yahoo	60.94%	38.03% (34.61%)	0.16%	0.86% (0.62%)

Table 10: Compositions of passwords that contain dates. The percentages outside parentheses are the ones counting both uppercase and lowercase letters, and the percentage inside parentheses are the ones counting only lowercase letters.

but most likely it is just two consecutive *123*. Thus, we selected 30 six-digit numbers that might cause such type of false positive [1]. Granted that we could have introduced false negatives or cannot manage to remove all the false positives for sure, these 30 numbers represent the patterns that have special meanings or are easy to remember, and most likely they do not map to any dates. For instance, '520520' has a similar sound as ' i love you i love you' in Chinese. Thus, we believe that eliminating them will increase the accuracy of our statistics.

Table 8 and Table 9 show the results. For example, the 29.24% in the first row in Table 8 means that among the 1,621,954 eight-digit numbers, 29.24% of them are in the format of *YYYYMMDD*. We can conclude that Chinese users prefer to use the format *YYYYMMDD* and *YYMMDD*. This conforms with Chinese conventions where people prefer to begin dates with years. On the contrary, a majority of English users prefer to end the date with years.

Password Composition. What are the compositions of passwords that contain dates? Are they composed of pure digits or mixed with letters? We calculated the percentages of digit-only, letter and digit, symbol and digit, letter and digit and symbol passwords out of all passwords that contain dates (both six-digit and eight-digit dates). As shown in Table 10, for all Chinese and English websites except Duduniu, most dates observed in our analysis are digit-only passwords, *i.e.*, when dates are used as passwords, they are used alone. What ranks the second is the passwords containing letters and digits. Note for Duduniu, the passwords that contain dates are more likely to contain both digits and letters than digits only. This could be because Duduniu is an e-commerce website and its users tend to choose a password with stronger strength, i.e., they tend to select passwords with both digits and letters, but not digits only.

[1]The 30 six-digit numbers are: *111111, 123123, 111000, 112233, 100200, 111222, 121212, 520520, 110110, 123000, 101010, 111333, 110120, 102030, 110119, 121314, 521125, 120120, 010203, 122333, 121121, 101101, 131211, 100100, 321123, 110112, 112211, 111112, 520521, 110111.*

	Beginning	Middle	End
CSDN	21.68%	4.32%	74.00%
Tianya	27.33%	4.75%	67.07%
Duduniu	24.76%	1.36%	73.88%
7k7k	32.17%	2.70%	65.13%
178.com	22.30%	1.03%	76.67%
RockYou	27.40%	3.91%	68.69%
Yahoo	22.66%	5.00%	72.34%

Table 11: Positions of dates. The percentages of passwords that contains dates at the beginning, the middle, or the end.

Date Position. To understand the position of dates in passwords, we analyzed those passwords that contain dates (digit-only passwords are not included).

We categorize the position of the dates as *beginning*, *middle*, and *end*, and summarize the results in Table 11. For both Chinese and English users, they prefer to have dates appear at the end of passwords and rarely place them in the middle.

2.3 Resistance to Guessing

Given the huge differences between Chinese and English passwords, a fundamental question is whether those differences lead to different levels of password strength. In this section, we examine password strength against password cracking.

2.3.1 Metrics to Measure Password Sets

We evaluated how resistant those passwords are against guessing by using the measurement metrics adopted by Bonneau [6][8], which are designed to evaluate the password strength in different regions.

As shown in Table 12, we briefly introduce these metrics: H_∞ is defined as *min-entropy*, a worst-case secu-

Metric	Formula	Term	Description
$H_\infty(\mathscr{X})$	$-\log_2(p_1)$		Worst-case security metric
$G(\mathscr{X})$	$\sum_{i=1}^{N} p_i \cdot i$	*guesswork*	The expected number of sequential guesses to find the password of an account if an attacker proceeds in optimal order
$\tilde{G}(\mathscr{X})$	$\log_2(2 \cdot G(\mathscr{X}) - 1)$		Bit representation of $G(\mathscr{X})$
$\mu_\alpha(\mathscr{X})$	$min\{j \in [1,N] \mid \sum_{i=1}^{j} p_i \geq \alpha\}$	α-*work-factor*	The expected number of guesses needed to succeed with probability α
$\tilde{\mu}_\alpha(\mathscr{X})$	$\log_2\left(\frac{\mu_\alpha(\mathscr{X})}{\lambda_{\mu_\alpha}}\right)$		Bit representation of $\mu_\alpha(\mathscr{X})$
$\lambda_\beta(\mathscr{X})$	$\sum_{i=1}^{\beta} p_i$	β-*success rate*	The probability that an attacker can correctly guess the password of an account given β guesses
$G_\alpha(\mathscr{X})$	$(1-\lambda_{\mu_\alpha}) \cdot \mu_\alpha + \sum_{i=1}^{\mu_\alpha} p_i \cdot i$	α-*guesswork*	The expected number of guesses per account to achieve a success rate α
$\tilde{G}_\alpha(\mathscr{X})$	$\log_2\left(\frac{2 \cdot G_\alpha(\mathscr{X})}{\lambda_{\mu_\alpha}} - 1\right) + \log_2\left(\frac{1}{2-\lambda_{\mu_\alpha}}\right)$		Bit representation of $\tilde{G}_\alpha(\mathscr{X})$

Table 12: Metrics [6][8] list used in our analysis. $\mathscr{X}$ refers to the probability distribution of passwords; N refers to the number of distinct passwords in a password set; p_i refers to the probability of the i-th password in $\mathscr{X}$ where $p_1 \geq p_2 \geq p_3 \geq \cdots \geq p_N$.

rity metric for human-chosen passwords, i.e., when a user chooses the mostly likely password. G is defined as *guesswork*, representing the expected number of sequential guesses to find a password of an account if an attacker proceeds in an optimal order, i.e., trying passwords in a descending order of the password probability. μ_α is called *marginal guesswork* or α-*work-factor*, which measures the expected number of guesses needed to succeed with probability α. *Marginal success rate* or β-*success rate*, λ_β, represents the probability that an attacker can correctly guess the password of an account given β guesses. G_α, the α-*guesswork*, reflects the expected number of guesses per account to achieve a success rate α.

To be more intuitive to programmers and cryptographers, we can convert these metrics into units of bits by taking the logarithmic value. We use a tilde over each letter to denote the values that are converted into bits: $\tilde{G}$, $\tilde{\mu}_\alpha$ and $\tilde{G}_\alpha$.

In this section, we follow the same assumption as proposed by Bonneau [6][8], *i.e.*, attackers know the exact distributions of the target password set and calculate the password strength, i.e., the attackers utilize the distribution of passwords to crack passwords in the same website. We call it *intra-site guessing*. In the next section, we relax the assumption, and we examine the guessing efficiency if the attackers are only aware of password distribution of other websites.

2.3.2 Resistance to Intra-Site Guessing

We summarize the calculated metrics for each website in Table 13 and Figure 3, and we draw the following observations:

- In Table 13, we observe that the β-*success-rates* (λ_5, λ_{10}) of `RockYou` and `Yahoo` are much lower than those of Chinese websites, i.e., given $\beta(e.g., 5, 10)$ guesses, the probability of guessing Chinese passwords correctly is higher. This phenomenon shows that Chinese websites have a lot of repeated passwords, but the $G_{0.25}$ and $G_{0.5}$ are similar (less than 3) between Chinese and English websites (except `178.com`). Thus, it may be easier to guess a small proportion of Chinese passwords, but for a majority of Chinese passwords, guessing them becomes as hard as guessing English ones.

- In Figure 3, the value of α-*work-factors* of `CSDN`, `Tianya` and `7k7k` are small if the expected success rate α is small, but it grows quickly with the increase of α. This phenomenon indicates that although part of Chinese users use the weak passwords that are easy to guess, a considerable number of users still carefully select passwords to protect their accounts. In addition, the users of `Duduniu` tend to choose better passwords. One possible explanation is that `Duduniu` involves monetary transaction and users tend to choose secure passwords.

	$\tilde{G}$	H_∞	λ_5	λ_{10}	$\tilde{G}_{0.25}$	$\tilde{G}_{0.5}$
CSDN	21.29	4.77	9.41%	10.44%	15.60	20.30
Tianya	21.49	4.55	7.15%	8.11%	14.67	19.11
Duduniu	22.55	6.02	2.74%	3.51%	18.94	21.59
7k7k	21.25	4.75	6.53%	7.61%	15.22	19.63
178.com	20.40	5.11	6.40%	8.74%	9.50	15.67
RockYou	22.65	6.81	1.71%	2.05%	15.88	19.80
Yahoo	18.03	8.05	0.78%	1.01%	16.31	17.68

Table 13: Resistance to guessing. H_∞ is the *min-entropy* for the most likely passwords. For $\tilde{G}$, H_∞, and $\tilde{G}_\alpha$, a larger value maps to stronger security. For λ_β, a smaller value indicates a lower possiblity of successful password cracking. Overall, the table shows that a small portion of Chinese passwords are repreaded and weak, but guessing a majority of Chinese passwords is as hard as guessing English ones.

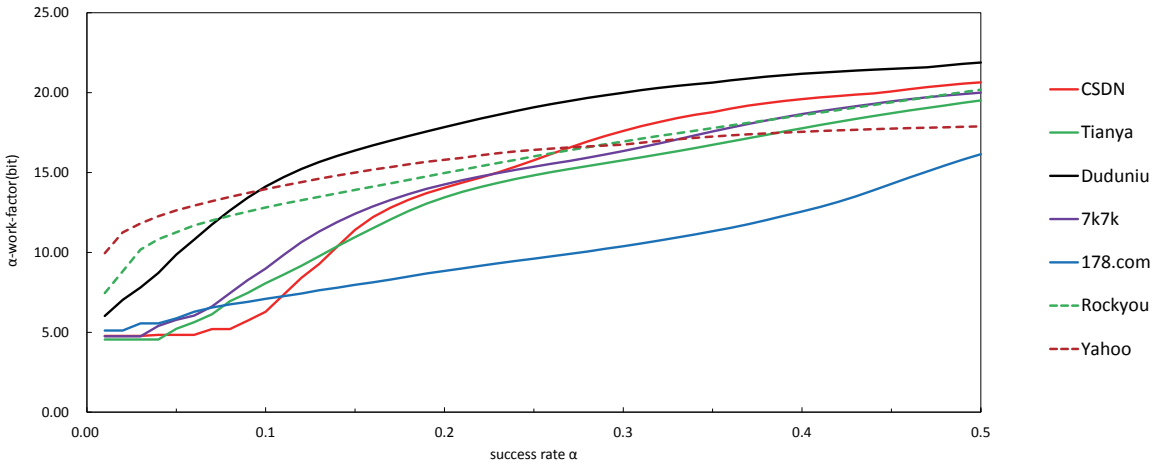


Figure 3: The expected number of guesses needed to succeed with a success rate α (α-work-factors, $\tilde{\mu}_\alpha$) of all seven websites. The dash lines represent English websites and solid lines map to Chinese websites.

3 Cross-Region Guessing

In this section, we would like to answer the following questions.

- Given that an attacker only has the password distribution of English websites, how well can she guess the passwords of Chinese websites?
- Given the knowledge of the differences between Chinese and English passwords, can an attacker improve the efficiency of guessing the passwords of Chinese websites?

The following two subsections answer these two questions.

3.1 Cross-Site Password Guessing

In this section, we examine how well an attacker can guess passwords from a website when she only possesses a password set of another website, and we call such scenarios as cross-site password guessing. This represents the situation when an attacker want to crack passwords of a website whose passwords have never been leaked. We modify the metrics that are modeled for the intra-website password guessing (listed in Table 12) to evaluate cross-site password guessing. We use two metrics, α-*work-factors* and β-*success-rates*, to evaluate the resistance to cross-site guessing. We denote these two metrics by adding a check symbol:

$$\check{\mu}_\alpha(\mathscr{X}) = min\{j \in [1, N_{other}] | \sum_{i=1}^{j} p_{(other)i} \geq \alpha\} \quad (1)$$

$$\check{\tilde{\mu}}_\alpha(\mathscr{X}) = \log_2\left(\frac{\check{\mu}_\alpha(\mathscr{X})}{\check{\lambda}_{\check{\mu}_\alpha}}\right) \quad (2)$$

$$\check{\lambda}_\beta(\mathscr{X}) = \sum_{i=1}^{\beta} p_{(other)i} \quad (3)$$

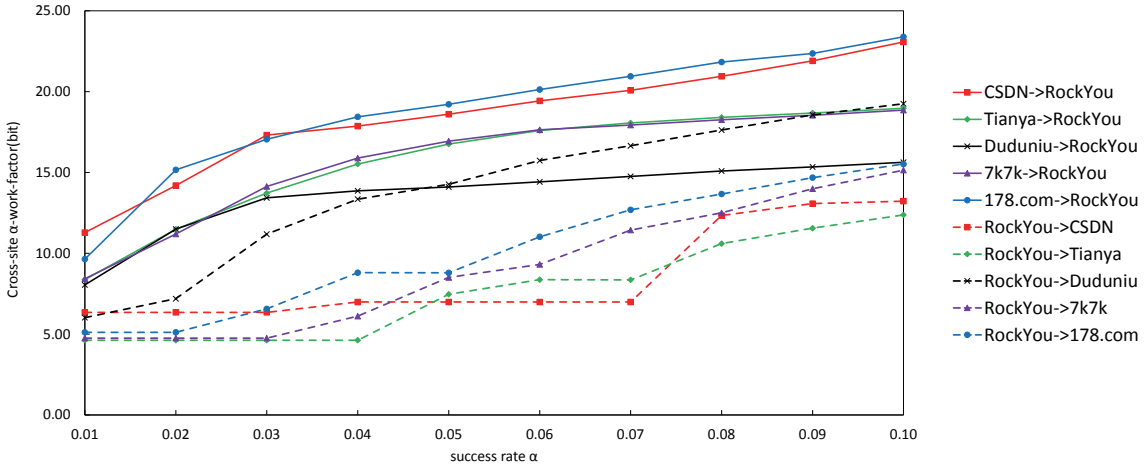


Figure 4: α-work-factors ($\tilde{\mu}_\alpha$) of cross-site guessing, i.e., the expected number of guesses needed to succeed with a success rate α. "X->Y" means using the X's optimal order to guess Y's passwords. For example, "CSDN->RockYou" means using the CSDN's optimal order to guess RockYou's passwords.

	Chinese Websites → RockYou		**RockYou → Chinese Websites**	
	$\check{\lambda}_5$	$\check{\lambda}_{10}$	$\check{\lambda}_5$	$\check{\lambda}_{10}$
CSDN	0.31%	0.35%	3.79%	7.11%
Tianya	1.24%	1.34%	4.78%	5.16%
Duduniu	1.18%	1.50%	2.11%	2.27%
7k7k	1.20%	1.28%	4.39%	4.66%
178.com	0.93%	1.00%	3.19%	3.33%

Table 14: β-success-rates of cross-site guessing. The data in columns 2 and 3 maps to the scenarios that we used each Chinese datesets to guess Rockyou passwords, and the data in columns 4 and 5 maps to the ones that we used Rockyou passwords to guess the ones of each Chinese website. These data shows that the cross-site guessing between Chinese and English users is hard.

In the above metrics, $p_{(other)i}$ refers to the probability of the other websites' i-th password in $\mathscr{X}$. For example, we utilize the CSDN's optimal password order to estimate the strength of Tianya's passwords, and $\mathscr{X}$ is the probability distribution of Tianya. In the CSDN's optimal order, "123456" is the first password. Given that in Tianya's passwords "123456" accounts for 0.52%, $p_{(CSDN)1}$ is 0.52%.

Using the methods mentioned above, we examine two scenarios: (1) given the passwords from the five Chinese websites as a prior knowledge, how well can we guess the passwords of RockYou; (2) given the passwords of RockYou, how well can we guess the passwords of the five Chinese websites. Note that we did not take Yahoo into consideration because of its small data size. The results of α-*work-factors* and β-*success-rates* of cross-site guessing are shown in Figure 4 and Table 14, where we can conclude that cross-site guessing is much harder than intra-site guessing (shown in Figure 3 and Table 13).

A lower β-*success-rates* means that the probability of correct guesses given β guesses are lower. In cases of using the information of Chinese passwords to guess the RockYou passwords, the β-success rates ($\check{\lambda}_5$ to $\check{\lambda}_{10}$) (listed in the 2nd and 3rd columns of Table 14) are lower than the intra-site guessing ones, i.e., $\lambda_5 = 1.71\%$ and $\lambda_{10} = 2.05\%$ for RockYou. In cases of using the information of the RockYou passwords to guess Chinese passwords, the β-success rates ($\check{\lambda}_5$ to $\check{\lambda}_{10}$) (listed in the 4th and 5th columns of Table 14) are also lower than the corresponding intra-site guessing listed in Table 13. A higher α-*work-factors* means that it takes a larger number of guesses to hit the right passwords. Compared with intra-site guessing (shown in Figure 3), for the same α value, the α-*work-factors* of the cross-site guessing (shown in Figure 4) is larger. Thus, cross-site guessing is harder.

Algorithm 1 Identify Keyboard Patterns

Input: S: a string
Output: the keyboard pattern of S

```
if S.length < 4 then
    return NO_PATTERN
end if
letters[] ← S.toCharArray()
samerow ← TRUE
zigzag ← TRUE
for i = 1; i < letters.length(); i++ do
    pos1 ← letters[i − 1]
    pos2 ← letters[i]
    if isAdjacent(pos1, pos2) then
        samerow ← samerow&isSamerow(pos1, pos2)
        zigzag ← zigzag&!isSamerow(pos1, pos2)
    else
        return NO_PATTERN
    end if
end for
if samerow then
    return SAME_ROW
end if
if zigzag then
    return ZIG_ZAG
end if
return SNAKE
```

Algorithm 2 Insert into the Trie

Input:
S: a string (a Chinese Pinyin or English word) that needs to be inserted into the Trie
Root: the root of the Trie

```
S ← S.toLowercase()
letters[] ← S.toCharArray()
node ← Root
for i = 0; i < letters.length(); i++ do
    pos ← letters[i] − 'a'
    node.child[pos].val ← letters[i]
    node ← node.child[pos]
end for
node.isValue ← TRUE
```

3.2 Guessing with Probabilistic Context-Free Grammar

The PCFG-based guessing method [35] increases the efficiency of password cracking process by trying passwords according to a decreasing order of password probability. The key of PCFG is to generate password rules (or structures). The rules can be constructed either from passwords themselves or word-mangling templates that can be filled in with dictionary words, for example. In our experiments, we built rules from three sources: (1) password sets, (2) dictionaries, and optionally (3) dates. We chose to use PCFG to examine whether the aforementioned rules are useful for guessing Chinese passwords, because it has been shown to be efficient in password guessing [21][24].

3.2.1 Methodology

We are interested in two questions: (1) How important are Pinyins and date formats for guessing Chinese passwords? (2) Given that an attacker is only aware of the English password distribution, can she synthesize a password distribution utilizing the differences that we have observed to improve the efficiency of cracking Chinese passwords?

To answer those questions, we created rules out of three types of sources for the PCFG-based guessing method: password training sets, dictionaries, and dates. For password training sets, we generated the following ones. Note that all training sets contain 2,000,000 passwords, respectively.

- **RockyouTS**: This training set contains passwords that are randomly chosen from `RockYou`. This represents a training set that only contains English password information.
- **MRockyouTS**: This training set also contains passwords from `RockYou`. However, the passwords are carefully selected so that its distribution follows the Chinese password distribution: 50% of the passwords are digit-only, and 10% are letter-only. This data set helps to examine whether the structure of passwords is enough to assist password guessing.
- **RockyouDuduTS**: Half of the passwords of this training set are randomly chosen from `Duduniu`, and the other half are randomly chosen from `RockYou`. This dataset helps to examine whether combined samples of Chinese and English passwords can assist password guessing.
- **DuduTS**: This training set contains passwords randomly chosen from `Duduniu` only. This represents the scenario that an attacker manages to obtain Chinese password sets.

In order to examine the effect of Pinyins in password guessing, we construct two dictionaries:

- **EDict**: This dictionary is a combination of the *Dic-0294* and *English-Lower*. *Dic-0294* is obtained

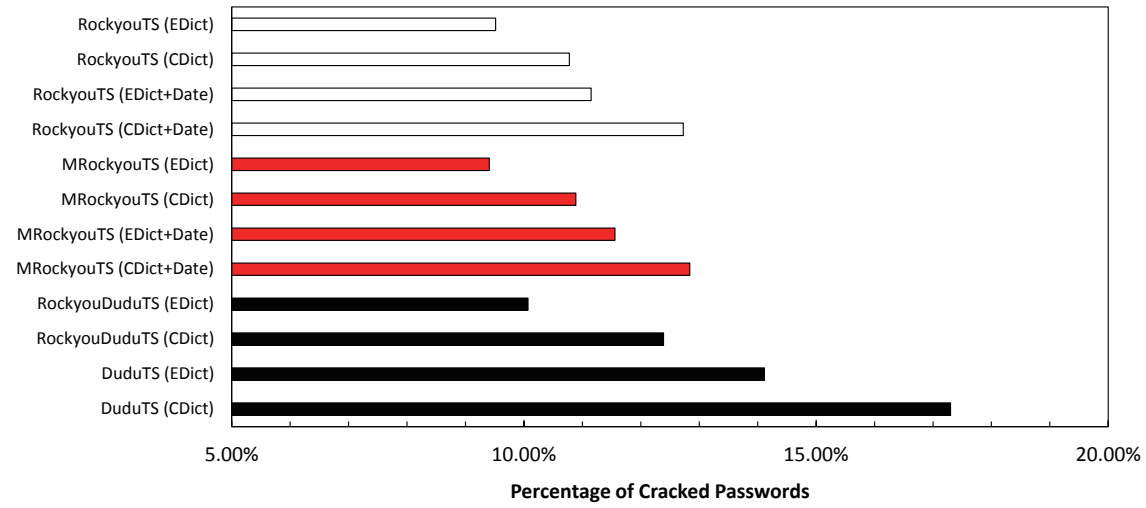


Figure 5: Passwords guessed within 10B guesses. Terminologies are explained in Section 3.2.1 in detail.

from a password guessing website [3] and *English-lower* is obtained from John the Ripper's public website [2]. *EDict* has 869,310 unique entries in total.

- **CDict**: To form this dictionary, in addition to *EDict*, we add 20,000 most frequently used Pinyins from the five Chinese websites. As a result, the size of *CDict* is larger than *EDict* by about 2.3%.

Besides Pinyins, dates also play an important role in password guessing. Since dates are digits, we modify the rules generated by the PCFG directly. We add 20,000 six-digit dates and 20,000 eight-digit dates that are most frequently used in the Chinese websites to the rules. These dates are assigned with the highest probabilities in the observed rules of six-digit numbers and eight-digit numbers, respectively. In total, these rules increase the number of six-digit and eight-digit rules by about 15% for *MRockyouTS* and about 31% for *RockyouTS*. We do not apply these rules to training sets *RockyouDuduTS* and *DuduTS*, because they already contain enough Chinese dates.

We used the above dictionaries and the modified rule set to guess the passwords of CSDN, and try 10 billion guesses per experiment.

3.2.2 Results of the PCFG based Guessing

As shown in Figure 5, the name of the training set is labeled on the left. In the parentheses, *EDict* and *CDict* represents which dictionary the guessing is based on and *Date* means that we added the dates to the rules generated by PCFG. According to Figure 5, we have the following conclusion.

- Chinese Pinyins and dates play an important role in guessing Chinese passwords. By adding 20,000 Pinyins into the dictionary, we managed to increase the percentage of password guessing. For *RockyouTS*, from *EDict* to *CDict*, the guessing efficiency increases by 13% and from *EDict+Date* to *CDict+Date*, the guessing efficiency increases by 14%.

  Furthermore, according to Section 2.2.3, more than half Chinese passwords are digit-only. For *RockyouTS*, the guessing efficiency increases by 17% after adding dates into *EDict* and it increases by 18% after adding dates into *CDict*. Last but not least, under the same dictionary, adding dates changes the percentage of guessed passwords of *RockyouTS* more than that of *RockyouDuduTS*.

- If we use the training set *RockyouTS* and *MRockyouTS*, the differences between the percentages of guessed passwords are small (less than 0.45% in all scenarios). This means that the distribution of password categories (e.g., letter-only, digit-only, etc.) does not play an important role in password guessing. It is the string patterns that make difference, since Chinese and English users prefer to use different patterns of digits and letters. Thus, using *RockyouDuduTS*, which consists both English password and Chinese password patterns can help the password guessing.

In total, from *EDict* to *CDict+Date*, we increase the guessing efficiency by 34% for *RockyouTS*. This guessing experiment imply that Pinyin and date's rules should be considered in password protection in websites. *E.g.*,

Algorithm 3 IdentifyComposition

Input:
S: a string that needs to match elements of Trie
Root: the root of the Trie

Output:
Whether the string S is composed of the element (s) in the Trie.

```
1: if S is NULL or S.length() is 0 then
2:    return FALSE
3: end if
4: letters[] ← S.toCharArray()
5: node ← Root
6: for i = 0; i < letters.length; i++ do
7:    pos ← letters[i]−'a'
8:    if node.child[pos] is NULL then
9:       if i is 0 then
10:         return FALSE
11:      end if
12:      if node.isValue is FALSE then
13:         return FALSE
14:      end if
15:      return IdentifyComposition(S.substring(i))
16:   else
17:      node ← node.child[pos]
18:      if IdentifyComposition(S.substring(i + 1)) is
         TRUE and node.isValue is TRUE then
19:         return TRUE
20:      end if
21:   end if
22: end for
23: return node.isValue
```

Web masters should tell Chinese users to reduce the usage of Pinyin or dates in composing their passwords.

4 Related Work

Although graphical passwords, biometrics and other alternatives to text-based passwords have been proposed, text-based passwords still predominate today's Internet due to its ease of implementation. A large body of research has shown the characteristics of user-created passwords [14][16][22][23][31][29][15].

Morris *et al.* [25] described the history of the design of the password security scheme and studied the password habits of 3,289 Unix users. Yan *et al.* [38] studied the password memorability and security. They found that users rarely choose passwords that are both hard to guess and easy to remember. Howe *et al.* [20] studied the behavior of home computer users because home computer users are more likely to suffer from various attacks, *e.g.*, phishing [36], dictionary attacks [27], heuristic password guessing [35], or brute force attacks. Florencio *et al.* [18] reported a large-scale study of Web passwords habits. The study involved half a million users over a three-month period. They found that on average, each user has 6.5 passwords and about 25 websites accounts. Kelly *et al.* [21] studied 12,000 actual passwords from several perspectives. They found that certain passwords policies which can improve the strength of user-created passwords are underestimated. In addition, a blacklist of weak passwords improves the security of passwords greatly. However, the aforementioned literature rarely mentioned the password difference between different regions, especially between Chinese and English users.

Bonneau [6] analyzed the language dependency of password guessing. The results show that among all `Yahoo` passwords, passwords created by Chinese are almost the hardest to guess. However, our experiments show that (1) the passwords of both English and Chinese users are similar in strength as shown in Figure 3 and Table 13; (2) if an attacker is aware of the fundamental differences between two languages (as pointed out in this paper), she or he can guess Chinese passwords efficiently. Moreover, our empirical study is based on two groups of websites: five Chinese websites, and two English websites, which represents a larger and more diverse corpus of passwords than `Yahoo` data set in Bonneau's work, and our corpus include passwords from users that only speak Chinese, unlike the Chinese users in Bonneau's work who should be familiar with English. Bonneau *et al*. [9] also investigated the lingering effects of character encoding on the password ecosystem based on password datasets from Chinese, English, Hebrew and Spanish speakers. Comparing with the results in [9], our large-scale empirical analysis in this paper also shows that the strength of the passwords of Chinese and English users is similar. Moreover, we firstly quantitatively measure how an attacker can leverage the lingering effects to crack more Chinese passwords.

In terms of measuring the strength of passwords, NIST standards [11] propose to use Shannon's entropy to estimate the strength of a single password. Unfortunately, this method does not work well. Bonneau [6][8] proposed a set of metrics to measure the strength of passwords. These metrics are independent of what the passwords are, but depend on the distribution of the passwords. We modify these metrics to estimate the strength of passwords across websites. In addition, Kelly *et al.* [21] used guess numbers to measure the strength of passwords.

Guessing passwords has attracted much attention. Narayanan *et al.* [26] discussed a password-guessing algorithm based on Markov model. In this model, guessing passwords is based on the frequency of each character. Weir *et al.* [35] proposed a PCFG based password guess-

ing method. The PCFG generates password structures in the highest probability order based on a training set of passwords. Then, it generates word-mangling rules and guesses passwords from these rules. This approach provides us with an opportunity to examine the differences between Chinese and English passwords. In addition, Veras *et al*. [34] employed Natural Language Processing techniques to understand the semantic patterns in passwords, then cracked more passwords than a state-of-the-art approach did.

5 Conclusion and Future Work

To the best of our knowledge, this paper is the first large-scale empirical study on Chinese Web passwords, leveraging a corpus of 100 million publicly available passwords. By comparing Chinese and English passwords, we find that Chinese users prefer digits in their passwords. Moreover, Pinyins and dates also appear often in their passwords. Leveraging these observations, we show that by adding rules and Pinyins into the dictionary for guessing passwords, we can improve the guessing efficiency of cracking Chinese passwords by 34%.

With an increasing number of password creation policies being enforced by websites, a direction for future study is to investigate the *status quo* of the password creation policies in Chinese websites and to study the impact of these policies on password statistics. Also, it is worthy exploring the semantic meanings of the Chinese passwords.

6 Acknowledgement

This paper is supported by Key Lab of Information Network Security, Ministry of Public Security (C13612), CNNIC DNSLab, Natural Science Foundation of Shanghai (12ZR1402600), 12th Five-Year National Development Foundation for Cryptography (MMJJ201301008), 1000 Young Talent plan from the China Central Organization, supported by the Fundamental Research Funds for the Central Universities (2013QNA4019). We also would like to thank Ari Juels for his suggestion and anonymous reviewers for their comments. Weili Han is the corresponding author.

References

[1] The concise oxford dictionray of current english. http://archive.org/stream/conciseoxforddic00fowlrich/conciseoxforddic00fowlrich_djvu.txt.

[2] Wordlist from john the ripper. http://download.openwall.net/pub/passwords/wordlists/.

[3] Wordlist from outpost9. http://www.outpost9.com/files/WordLists.html.

[4] 178.COM. http://www.178.com/s/information/about.html.

[5] 7K7K. http://www.7k7k.com/html/about.htm.

[6] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proceedings of 2012 IEEE Symposium on Security and Privacy (SP)* (2012), pp. 538–552.

[7] BONNEAU, J., HERLEY, C., VAN OORSCHOT, P. C., AND STAJANO, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of 2012 IEEE Symposium on Security and Privacy (SP)* (2012), IEEE, pp. 553–567.

[8] BONNEAU, J., PREIBUSCH, S., AND ANDERSON, R. A birthday present every eleven wallets? the security of customer-chosen banking pins. In *Proceedings of the 16th International Conference on Financial Cryptography (FC '12)* (2012).

[9] BONNEAU, J., AND XU, R. "of contraseñas, sysmawt, and mìmă: Character encoding issues for web passwords". In *Web 2.0 Security & Privacy* (May 2012).

[10] BOZTAS, S. Entropies, guessing, and cryptography. Tech. rep., Department of Mathematics, Royal Melbourne Institute of Technology, 1999.

[11] BURR, W. E., DODSON, D. F., NEWTON, E. M., PERLNER, R. A., POLK, W. T., GUPTA, S., AND NABBUS, E. A. Nist special publication 800-63-1 electronic authentication guideline, 2006.

[12] CNNIC. The 33rd survey report on chinese internet development. http://www.cnnic.cn/hlwfzyj/hlwxzbg/hlwtjbg/201401/t20140116_43820.htm, Jan 2014.

[13] CSDN. http://www.csdn.net/company/about.html.

[14] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *Proceedings of NDSS 2014* (2014).

[15] DE CARN DE CARNAVALET, X., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Proceedings of NDSS 2014* (2014).

[16] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *Proceedings IEEE INFOCOM 2010* (2010), IEEE, pp. 1–9.

[17] DUDUNIU. http://baike.baidu.com/view/1557125.htm.

[18] FLORENCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)* (2007), pp. 657–666.

[19] HERLEY, C., AND VAN OORSCHOT, P. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy 10*, 1 (2012), 28–36.

[20] HOWE, A., RAY, I., ROBERTS, M., URBANSKA, M., AND BYRNE, Z. The psychology of security for the home computer user. In *Proceedings of 2012 IEEE Symposium on Security and Privacy (SP)* (2012), pp. 209–223.

[21] KELLEY, P., KOMANDURI, S., MAZUREK, M., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proceedings of 2012 IEEE Symposium on Security and Privacy (SP)* (2012), pp. 523–537.

[22] KUO, C., ROMANOSKY, S., AND CRANOR, L. F. Human selection of mnemonic phrase-based passwords. In *Proceedings of the Second Symposium on Usable privacy and security* (2006), ACM, pp. 67–78.

[23] MALONE, D., AND MAHER, K. Investigating the distribution of password choices. In *Proceedings of the 21st International Conference on World Wide Web* (2012), ACM, pp. 301–310.

[24] MAZUREK, M. L., KOMANDURI, S., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., KELLEY, P. G., SHAY, R., AND UR, B. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security* (2013), ACM, pp. 173–186.

[25] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Communications of the ACM 22*, 11 (1979), 594–597.

[26] NARAYANAN, A., AND SHMATIKOV, V. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), ACM, pp. 364–372.

[27] PINKAS, B., AND SANDER, T. Securing passwords against dictionary attacks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (2002), ACM, pp. 161–170.

[28] PLIAM, J. O. On the incomparability of entropy and marginal guesswork in brute-force attacks. In *INDOCRYPT* (2000), pp. 67–79.

[29] R. VERAS, C. COLLINS, J. T. On the semantic patterns of passwords and their security impact. In *Proceedings of NDSS 2014* (2014).

[30] ROCKYOU. `http://rockyou.com/ry/about-us`.

[31] SAWYER, D. A. The characteristics of user-generated passwords. Tech. rep., DTIC Document, 1990.

[32] SCHWEITZER, D., BOLENG, J., HUGHES, C., AND MURPHY, L. Visualizing keyboard pattern passwords. In *Proceedings of 6th International Workshop on Visualization for Cyber Security (VizSec 2009)* (2009), IEEE, pp. 69–73.

[33] TIANYA. `http://help.tianya.cn/about/history/2011/06/02/166666.shtml`.

[34] VERAS, R., COLLINS, C., AND THORPE, J. On the semantic patterns of passwords and their security impact. In *Proceedings of NDSS 2014* (2014).

[35] WEIR, M., AGGARWAL, S., DE MEDEIROS, B., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 391–405.

[36] XIANG, G., AND HONG, J. I. A hybrid phish detection approach by identity discovery and keywords retrieval. In *Proceedings of the 18th Internation Conference on World Wide Web (WWW'09)* (2009), pp. 561–570.

[37] YAHOO. `http://info.yahoo.com/`.

[38] YAN, J., BLACKWELL, A., ANDERSON, R., AND GRANT, A. Password memorability and security: empirical results. *IEEE Security Privacy 2*, 5 (2004), 25–31.

A Method to Remove the Copied Passwords in Tianya

`Tianya` and `7k7k` have an unusually large number of the same accounts (identified by email) with the same passwords. Since the statistic features of these duplicated accounts are different from the ones between any other websites, thus we suspected that the attackers have copied accounts from `Tianya` to `7k7k` or vice versa.

To investigate whether the accounts are copied from `Tianya` to `7k7k` or vice versa, we performed the following analysis. We first divided all accounts from `Tianya` and `7k7k` into two groups: One group contains the users who have the same accounts and passwords both at `Tianya` and `7k7k`, and the other contains the users who do not. We call the passwords of the two groups *reused passwords* and *not-reused passwords*.

After analyzing the compositions (e.g., digit-only passwords) of the *reused passwords* and *not-reused passwords*, we found that the proportions of various compositions are similar between the *reused passwords* and the `7k7k`'s *not-reused passwords*, but different with *Tianya*'s *not-reused passwords*. As a result, we believe that it is likely that accounts have been copied from `7k7k` to `Tianya` and we deleted the *reused passwords* from `Tianya`.

Password Portfolios and the Finite-Effort User: Sustainably Managing Large Numbers of Accounts*

Dinei Florêncio and Cormac Herley
Microsoft Research, Redmond, USA

Paul C. van Oorschot
Carleton University, Ottawa, Canada

Abstract. We explore how to manage a portfolio of passwords. We review why mandating exclusively strong passwords with no re-use gives users an impossible task as portfolio size grows. We find that approaches justified by loss-minimization alone, and those that ignore important attack vectors (e.g., vectors exploiting re-use), are amenable to analysis but unrealistic. In contrast, we propose, model and analyze portfolio management under a realistic attack suite, with an objective function costing both loss and user effort. Our findings directly challenge accepted wisdom and conventional advice. We find, for example, that a portfolio strategy ruling out weak passwords or password re-use is sub-optimal. We give an optimal solution for how to group accounts for re-use, and model-based principles for portfolio management.

1 Introduction

Due to the growth in online services, many users now manage dozens of password-protected accounts. Many service providers, awareness campaigns (US DHS [1]), and government entities (US-CERT [2]) stress two foundations for password security:

A1: Passwords should be random and strong; and

A2: Passwords should not be re-used across accounts.

Despite this, users have long been observed to choose weak passwords. Leaked datasets, such as the 32 million plaintext passwords from Rockyou, reveal that most users fall far short of following "traditional" advice on password strength. Evidence also indicates widespread password re-use [21]. While admonitions against this are almost universal, ignoring that advice seems equally universal. Clearly, users find managing a large password portfolio burdensome. Both password re-use, and choosing weak passwords, remain popular coping strategies.

Numerous efforts have been made to address the neglect of password strength by users. Many sites stress the importance of, and offer tips on how strong passwords can be made easier to construct and remember; e.g., US-CERT [2] and others commonly suggest passphrase-based and other mnemonic approaches. But while significant attention has been devoted to motivating and helping users choose strong individual passwords, there is little guidance on how to choose and manage large numbers of them. We aim to give, and justify, such guidance.

We explore how a large portfolio of passwords can be maintained without ignoring that users have limited abilities. Can password re-use be part of sensible portfolio management, or is it never justifiable? Is a unique strong password for every account, including blog sites and throw-away accounts, truly the best use of limited human memory resources? In practice, many users gather accounts into groups that re-use a password, but little guidance exists on choosing appropriate groups. Given that re-use does and will happen, we explore how to do so in a principled way, and answer these questions.

Our findings directly challenge some conventional wisdom. For example, we find: *strategies that rule out password re-use or the use of weak passwords are sub-optimal*. Both are valuable tools in balancing the allocation of effort between higher and lower value accounts.

We first review password-related demands on users, and consider users' options under the reasonable but too-rare assumption of *finite user effort*. This realism yields an inherent trade-off between two desired outcomes: greater password strength and avoiding re-use. Acknowledging fixed user effort budgets, more of one means less of the other.

We explore the implications of password re-use, and outline an optimal password-sharing strategy: for a fixed number of passwords and a given set of accounts, how to partition accounts to minimize total expected loss. Loss analysis is greatly complicated by cross-contamination issues due to password re-use. We address this by a novel

*USENIX Security 2014, August 20-22.

partitioning of attacks into three broad classes covering the major threat vectors, itself of independent interest.

2 Related Work

In 2000, Dhamija and Perrig [18] interviewed 30 participants reporting 1–7 unique passwords for 10–50 web sites. Circa 2001, Sasse and Brostoff [45] surveyed 144 employees reporting on average 16 passwords including non-online activity. A 2004 survey of 218 college students by Brown et al. [11] indicated on average 8.18 password accounts serviced by 4.45 unique passwords. In 2006 Gaw and Felten [24] surveyed 58 (mainly student) participants by online questionnaire with in-lab follow-up of 49, exploring how users manage online passwords, the extent of reuse and how users justify it, and the use of related passwords; they reported on average 13 passwords and found reuse increased over time—new accounts accumulated faster than new passwords. Riley's 2006 survey [44] of 315 college students (8.5 accounts on average) reported: 74.9% have a set of predetermined passwords they frequently re-use; 54.6% very frequently or always use a same password for multiple accounts; 33% use some variation of a same password for multiple accounts; and 60% do not vary the complexity of their passwords with the nature of a site. In a 2007 study of password use/re-use across three months by over a half million users, Florêncio and Herley [21] reported on average 25 accounts serviced by 6.5 unique passwords, re-used passwords used on average at 5.7 sites, and strong passwords re-used less.

Notoatmodo's 2007 thesis [42] explored password re-use and users' perspectives of their real-world passwords—and especially relevant to our work, how users mentally group both accounts and passwords into categories, relationships between account and password groups, and details of users' reasons both for, and for not, reusing passwords. The 26 participants surveyed had on average 12.9 accounts and 8.1 passwords; most reused passwords (132 of 336 accounts had unique passwords). Reuse was found again (see above) to increase with number of accounts. A hypothesis progressed was that users manage their accounts and passwords by mentally separating both into categories based on perceived account similarities and password similarities,[1] Regarding grouping accounts, and which accounts they felt were "high importance", most participants had only one high importance account group (1.54 such groups on average), and high importance groups were found to be smaller (fewer accounts per group: mean 1.84 vs. 2.78 for low importance groups). 45% reported reusing at least one password from a high importance group vs. 96% reusing at least one password from a low importance group; 70% had passwords exclusively used for an account in high importance groups (2.9 such passwords on average). In line with our views, Notoatmodo suggests *"reusing passwords on unimportant accounts which contain no sensitive information should not be discouraged ... Expecting users to create unique, strong passwords for all their accounts is ... unreasonable ... Instead, users should be educated to identify which accounts [not to] reuse passwords on."* While granting that password re-use is dangerous, Karp [36] also argues for re-use (*"human nature being what it is, not reusing passwords is equally dangerous"*) but in a different direction: by a password manager tool re-using a user password as a master password combined with details of a target site (e.g., site name) for site-specific passwords.

The "domino effect" of password re-use is well-documented (e.g., Ives et al. [32]; Gouda et al. [25]). The need for re-use is exacerbated by large numbers of passwords consuming user's memory capacity [3]. In scarce empirical work on implications of password re-use, Bonneau and Preibusch [9] analyze password implementations across 150 free websites, explaining technical means by which password re-use allows low-security sites—often unmotivated to spend effort or user experience securing passwords—to compromise high-security sites. The same authors [43] explore this question as a negative externality of password policies, finding a tragedy of the commons whereby sites with the lowest security needs can endanger those with the highest. Florêncio and Herley [22] find that the imposition of stringent password policies is better correlated with insulation from the consequences of poor usability than the need for greater security.

While the degree of password re-use naturally varies with the users studied, their circumstances and environment at a give time, evidence clearly shows it is widespread. The accuracy of self-reported re-use statistics is debatable, but a lower bound on re-use in real life is possible from leaked password databases from two different sites: from each database, recover a (userid, password) list, find userids common to both (e.g., re-used email addresses), then count password re-use instances. Das *et al.* [17] estimate that 43-51% of users re-use passwords across sites, and give algorithms that improve an attacker's ability to exploit this fact; this exceeds the 12-20% rate of some earlier studies noted above [24, 21]. Lemos [38] reports that intersection of the breached database pair (Yahoo Voices, Sony online) and (Sony online, Gawker) found usernames had re-used passwords across two sites 59% and two-thirds of the

[1] Examples of similarities for grouping passwords: "school stuff", email accounts, online banking, and semantic properties related to security (e.g., overall length, number of letters). Examples for account grouping: type of service related to the account (e.g., financial, education, communication), similar levels of risk or importance.

time in the two pairs. RockYou's leaked dataset [30] was explored by Bonneau [7, p.83] and Weir et al. [48]. Zhang et al. [49] easily predict new passwords from old when password aging policies force updates.

Over a 2011 two-week diary study of password use by Hayashi et al. [28], 20 participants reported 8.6 accounts on average, and to not use any memory aids for 60% of accounts; 19 of 20 said they reused passwords for multiple accounts. This may indicate under-estimating password re-use risk vs. writing passwords down. From a 2010 one-week diary-based study wherein 32 staff from two organizations produced just over 6 passwords each, Inglesant et al. [31] suggest that password polices be designed not to maximize password strength but rather to aid users in setting strengths appropriate to specific use contexts. Grawemeyer et al. [26] explore re-use among coping strategies in managing collections of passwords, in a detailed 2011 diary study of 22 participants over 7 days. In 2014, Stobert et al. [47] also explore user coping strategies for managing passwords, with guided interviews and questionnaires on 27 participants—noting as a user concern "rationing effort to best protect important accounts", and that "many participants [reported] having a specific password that they reused widely on accounts of low interest, low importance, or infrequent use".

The idea of grouping passwords, e.g., by level of importance, has seen little academic study, but Cheswick et al. [15, pp.140-141] suggested four categories: worthless, slightly important, quite secure, and top security. Cheswick more recently [13] suggests three classes: those that (a) have no importance; (b) are inconvenient if stolen; or (c) result in a major problem if abused. Five categories each are given by Grosse et al. [27] (based on account value) and Florêncio et al. [23] (based on consequence of compromise). Cheswick [14] also reviews common password guidance, and the ongoing suitability of circa-1985 U.S. government password guidelines. Florêncio and Herley suggest defender goals are well modelled by minimizing loss plus effort [20].

Nithyanand et al. [41] explore issues related to password re-use, under an attack model focused on server-side breakin (excluding phishing, client-side malware); seek solutions to maximize "remaining value" (i.e., minimize loss, vs. loss plus effort herein); show their password allocation problem is NP-complete; and find heuristic solutions to special cases (for accounts of equal value, with identical compromise probabilities, etc.).

3 The Difficulty of Managing a Portfolio

Issues related to complexities of human memory, encoding and recalling information (see [10]) currently preclude a satisfactory cognitive model or measure of the load passwords place on users. Nonetheless we begin with a naive model to highlight impossible-to-meet assumptions, and to position and motivate later discussion. We stress that *our later modeling* (Sections 4 onward) *abandons these assumptions and this naive model*, tackling a more realistic setting. We acknowledge that our equations in this Section, e.g., for the difficulty of associating passwords with accounts, give at best crude estimates. We emphasize also that this paper considers ordinary text passwords, not text or graphical variations using cues; we make no claims regarding such schemes.

An active web-user may have a hundred or more password-protected accounts. Ideally a user with N accounts chooses N strong passwords. If passwords were random collections of equi-probable characters, the difficulty of remembering them would be related to their length. Assume each such password is $\lg S$ bits. The effort required to manage the portfolio might naively appear to be $N \lg S$. But beyond remembering N passwords, users must remember which matches which account. We now explicitly consider this often overlooked sub-task.

3.1 Matching Passwords to Accounts

There are $N \cdot (N-1) \cdots 1 = N!$ possible mappings of N unique passwords to accounts; no encoding of this information uses less than $\lg(N!)$ bits, unless passwords contain clues as to which site they serve, violating A1 above. Thus the number of bits to be remembered to manage a portfolio of N passwords, each of $\lg S$ bits, is at least:

$$E(N) = N \cdot \lg S + \lg(N!). \tag{1}$$

(As noted above, this approximation fails to address the complexities of human cognition, but suffices for the argument below.) Clearly this grows rapidly with N, the second term super-linearly by Stirling's approximation ($\ln N! \approx N \ln N - N$). Consider a conscientious user, with $N = 100$ accounts. Choosing unique random passwords of 40 bits for each account rewards him with the obligation to remember $100 \times 40 + \lg(100!) = 4525$ bits (equivalent to 1362 random digits or 170 random 8-digit PINs). This burden far exceeds what users can manage by memorization (*i.e.*, without other aids); for most it is insupportable. How can users reduce it? An obvious shortcut, with significant side effect, is to choose weaker (less random) passwords; the linear dependence on $\lg S$ suggests reducing strength as much as possible.

While using weaker passwords clearly reduces the first term of (1), no matter how weak N distinct passwords are, the second term is unaffected. Considering that term alone, $N = 100$ yields $\lg(N!) = 525$. This is double the $\lg(52!) = 226$ bits required to memorize the order of a shuffled card deck, and equivalent to remembering 158 random digits—random since as noted, no encoding of an $N \times N$ assignment takes fewer than $\lg(N!)$ bits. Thus,

the assignment burden alone, including the problem of *password interference* [16], is evidently beyond a reasonable expectation of users.

So the two staples A1, A2 of password advice appear impossible to meet individually, let alone simultaneously. How do users proceed? They "cheat" on A1 by choosing passwords far weaker than advised. But this isn't enough—no matter how weak the passwords, a user must still remember $\lg(N!)$ random bits for password assignment. A further coping strategy is needed.

3.2 Password Re-use as a Coping Strategy

Consider next a user with N accounts using $G \leq N$ passwords to cover them. Assume for now each password is used at $n = N/G$ accounts, that the password-to-group assignment is random and (for simplicity) that G divides N. The burden of remembering passwords drops to $G \cdot \lg S$ bits. What of the further burden of remembering which password goes where? The G groups of accounts each have $n = N/G$ elements. There are C_n^N possible combinations for the first group, C_n^{N-n} for the second, etc., so the number of possible assignments of N accounts to G equal-sized groups is:

$$\left(\begin{array}{c} N \\ n \end{array}\right) \cdot \left(\begin{array}{c} N-n \\ n \end{array}\right) \cdots \left(\begin{array}{c} n \\ n \end{array}\right) = \frac{N!}{(n!)^G}.$$

Thus the user effort (memory burden in bits) drops to:

$$\begin{aligned} E_G(N) &= G\lg S + \lg(N!) - G \cdot \lg(n!) \qquad (2) \\ &\approx G\lg S + N\lg G \end{aligned}$$

the last line following by Stirling's approximation again.

Now compare the burden of managing a portfolio with and without password re-use. For example, even if $\lg S$ is as low as 20, from (2), the burden of managing 100 accounts with 10 passwords is $E_{10}(100) = 506$ bits, while from (1) the burden of doing so with 100 is $E(100) = 2525$ bits. Thus, in this instance, password re-use reduces the memorization burden by a factor of five.

3.3 Tradeoff: Re-use & Password Strength

What other solutions use the same effort? A portfolio of N passwords can be managed in many ways. If $E_G(N)$ is fixed then $\lg S \approx (E_G(N) - N\lg G)/G$. So, $\lg S$ falls faster than $1/G$: doubling the number of passwords more than halves the number of bits per password. So, if $G = N$ (no password re-use), then $\lg S$ must be small.

Fig.1 shows the locus of solutions in the G-$\lg S$ plane when $N = 100$ and the budget is $E_G(100) = 400, 550$ and 700 bits. This reveals the essential tradeoff: less re-use (*i.e.*, increasing G) implies weaker passwords. For example, at fixed effort $E_G(N) = 400$, two possible operating points are $(G = 4, \lg S = 52.5)$ and $(G = 5, \lg S = 36.2)$. At fixed effort, the question is not whether achieving password strength and avoiding re-use are good, but how these relative goods are best traded off. Deciding, e.g., between these two operating points depends on whether reducing password re-use (by increasing G from 4 to 5) reduces the risk of harm by more or less than reducing password strength (from 52.5 to 36.2 bits). The rapid decline of $\lg S$ in Fig.1 as G increases suggests that, far from being unallowable, password re-use is a necessary and sensible tool in managing a portfolio. Re-use appears unavoidable if $\lg S$ must remain above some minimum (and effort below some maximum). Fig.1 further suggests that G should be small: high values of G seem to imply very low values of $\lg S$. This enormous saving in user effort that password re-use provides may explain its ongoing prevalence in practice [24, 19, 21, 45].

Note on entropy: caution is needed to avoid historical pitfalls such as assuming particular ranges for $\lg S$ based on metrics appropriate only for *random* passwords, or misleading rules-of-thumb on what is necessary to withstand attack. We intend $\lg S$ to represent user effort to remember a password, not attacker guessing difficulty; the two may be correlated but should not be used in place of each other. For example, if a user has 7 unique passwords, and each names a major Hawaiian island, then $\lg S = \lg 7 \approx 2.8$ bits. But this offers little guide on how hard these passwords are to guess. It is also well understood [48, 8, 37, 39] that $t \cdot \lg C$ and NIST's crude password entropy estimate [12], significantly overestimate the difficulty of guessing *user-chosen* length-t passwords from C-character alphabets. Equally, such metrics must not be mis-used to estimate users' capabilities or effort, lest we drastically over-estimate what a reasonable cognitive burden is.

4 Objective Function: Loss + Effort

Suppose a user has N password-protected accounts. Advice such as A1, A2 implicitly assume the goal is to minimize loss. Let P_i be the probability of compromise in a given period (e.g., per year, under the current password management strategy), and L_i the loss endured upon such compromise. We intend that P_i capture the probability that an attacker gains the means to access account i, whether or not that means is used or results in loss. We intend L_i to capture the *expected value* of the consequences of account compromise (regardless of attack vector), including direct losses and any indirect costs involved in remediation. The total expected loss is

$$L = \sum_{i=1}^{N} P_i \cdot L_i. \qquad (3)$$

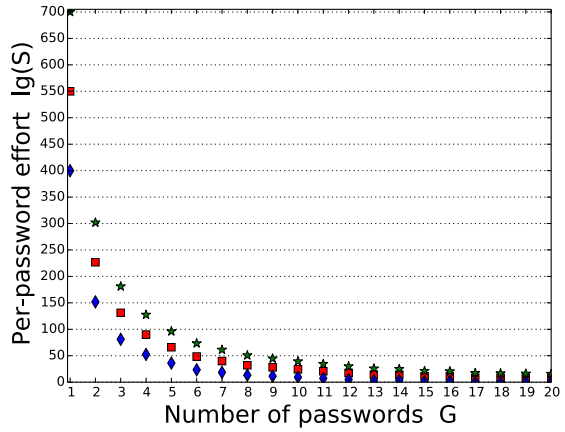


Figure 1: Locus of achievable solutions trading off re-use and password strength for fixed effort $E_G(N) = 400,550,700$ in (2) at $N = 100$. Note that when effort is kept constant, lower levels of re-use are only achieved by having weaker passwords.

A major complication we will find—and defer to Section 5—is that some attacks affect more than one account; e.g., malware and system attacks affect all accounts, and if passwords are re-used then an attack against one account can affect many others. But for now, suppose that attacks are only against individual accounts. Then the P_i depend on effort E_i devoted to account i but not to E_j. The probability of compromise $P_i = P_i(E_i)$ is presumably monotonically non-increasing with effort. Investing more effort generally reduces P_i; a stronger password reduces the risk that it falls to password-guessing attacks.

If aiming to minimize expected loss L, the solution occurs when L has derivative zero with respect to $E = \sum_i E_i$, which from (3) gives the system of equations

$$\frac{dP_i}{dE_i} = 0 \quad \text{for} \quad i = 1,2,3,\cdots,N. \tag{4}$$

The solution is trivial: the optimum is achieved when further effort can't reduce the probability of loss for any of the N accounts. Thus to minimize L, we should increase each E_i until no further reduction is possible (further effort does not affect P_i). If $P_i(E_i)$ is monotonically decreasing—so further effort always reduces P_i—then expected loss is minimized at infinite effort. Thus *the lack of a constraint on effort leads to an unrealistic solution.* Note also that users gravitate toward a solution very different from this effort-maximizing one. From this we infer: *the objective function users minimize is not merely expected loss*—if it were they'd always invest effort that could reduce loss and always follow A1, A2.

This optimization has an obvious flaw: minimizing L implicitly values user effort at zero. Users presumably care about loss due to account compromise, *but they also factor in the effort they must spend to reduce that loss.* They may be willing to spend effort to reduce loss, but at some point there are diminishing returns; and it is wasteful to continue after the cost of further effort exceeds expected reduction in loss. Thus, rather than an unconstrained optimization [29] ignoring reluctance to increase effort, we should solve a constrained problem explicitly including cost of user effort.

One way to incorporate a constraint is to minimize loss subject to a bound—e.g., say $\sum E_i < E_{max}$ to model users with an upper limit on the effort they are willing to exert. This is reminiscent of the compliance budget [5, 4]; it can be achieved with Lagrangian multipliers. A more general approach, which we follow, is to minimize not the expected loss, but the sum of effort plus loss, $L+E$. As a precedent for this approach, economics Nobel laureate Becker notes [6] that attempts to minimize crime lead to perverse results, and it is preferable to minimize the costs of crime plus the costs of detecting, prosecuting and punishing it. For example, it makes little sense to spend $1 more on policing effort if that reduces the effects of crime by less than $1.

To illustrate the importance of the objective function we revisit the question of finding optimum allocation of effort[2] when $dP_i/dE_j = 0, i \neq j$ (*i.e.*, no cross-account attacks). The optimum occurs when the derivative (with respect to E) of objective function $L+E$ is 0: $dL/dE + 1 = 0$. Using (3) to substitute for L gives the system

$$L_i \cdot \frac{dP_i}{dE_i} = -1 \quad \text{for } i = 1,2,3,\cdots,N. \tag{5}$$

Thus, at optimum effort allocation,[3] the marginal return on effort to reduce P_j is a factor L_i/L_j higher than that to reduce P_i :

$$\frac{dP_j}{dE_j} = \frac{L_i}{L_j} \cdot \frac{dP_i}{dE_i}. \tag{6}$$

If the loss for the most important account is, say, 10^4 times that for the least important ($L_1 = 10^4 L_N$) then the marginal return on effort should differ by that factor. Thus, *effort should not be spent equally on all accounts*.

While we are unlikely to find an exact form for how the probability of harm varies with effort, using a parametric form can help illustrate the relations. Basing an example on Shamir's quote "to halve your vulnerability you have to double your expenditure" [46], we examine what happens when there is a reciprocal relation between them, *i.e.*, $P_i(E_i) \propto 1/E_i$. This gives $dP(E_i)/dE_i \propto -1/E_i^2$. Substituting into (6) indicates how the relative effort for two accounts should depend on the relative losses:

$$E_j = \sqrt{\frac{L_j}{L_i}} \cdot E_i.$$

[2] This includes max cumulative effort and where/how to allocate it.
[3] This is also true at other points, though not proven here.

So if two accounts differ in value by factor 10^4, ideally the effort expended would differ by a factor 100. We reiterate: this analysis, as it depends on the parametric form for $P_i(E_i)$, is for illustrative purposes only.

Now contrast this solution minimizing $L+E$, with that found by minimizing L alone (system (4) above). First, if minimizing L, all passwords should be as strong as possible, meaning that (at the optimum) no additional effort can reduce the risk for any account. When minimizing $L+E$ this isn't the case: (5) says that (at the optimum) additional effort may still reduce risk for every account, but it is sub-optimal to spend it. Second, when minimizing L, the optimum protection given to an account is independent of L_i. When minimizing $L+E$ some accounts should be (possibly far) less protected than others: (5) shows that the rate of return on effort should be *inversely related to the account value*.

Thus, using objective function $L+E$ (not L) makes an enormous difference in solutions. We posit that much of the advice directed at users aims to minimize L only, and is ignored as users implicitly care about E also and have found operating points attempting to minimize *their* objective function; these points may or may not be optimal, but have been arrived at by *ad hoc* methods. We note that in minimizing $L+E$ we neglect the non-linear response to probabilities predicted by Prospect Theory [35]. We believe that the rational model which offers (Kahneman [34]) "great precision in some situations and good approximation in many others" is the most realistic one that we can currently make progress on, and significantly advances a model that neglects E. Finally, use of the term *portfolio* is not accidental. Since 1952 [40] it has been recognized that managing a portfolio of equities raises issues drastically different from managing individual securities. In an analogous situation for passwords, due to cross-account attacks, the security of accounts cannot be considered in isolation, yet the literature has given little attention to the portfolio problem.

5 Modeling Loss, Effort, Attack Classes

While (5) offers to guide effort allocation when minimizing $L+E$, it assumed $dP_i/dE_j = 0$ for $i \neq j$; we postponed issues of cross-account attacks. This might be reasonable if guessing were the only attack and account passwords were unique; the probability P_i of compromise of account i would then depend only on how passwords withstood attack. But that over-simplifies. With password re-use, compromise of one account can leak to others, and client-side malware affects all accounts. Such attack vectors are too important to ignore. P_i depends on effort not just devoted to account i but also, e.g., to address client malware or avoid phishing, and the security of other sites the password is re-used on.

If we can't assume partial derivatives of zero, then on minimizing $L+E$, instead of (5) we get the system

$$\sum_{i=1}^{N} \frac{\partial P_i}{\partial E_j} \cdot L_i = -1 \quad \text{for } j = 1,2,3,\cdots,N. \qquad (7)$$

This is not simply a linear system. The N unknowns E_j, specified implicitly by the constraint on N^2 partial derivatives, relate non-linearly to the L_i. The intuition of (5) is now lost. A simple interpretation (e.g., marginal return on effort should be inversely related to loss) is no longer discernible, as instead of appearing singly, the partial derivatives are now constrained by a sum.

Note that if we minimize L instead of $L+E$ we get a system similar to (7), but with zero on the right side. Since losses must be non-negative and the partial derivatives are non-positive, the solution is achieved by setting $\partial P_i/\partial E_j = 0$ for all i, j. This would again indicate optimality occurs when no further effort can reduce any of the loss probabilities. Thus, the fully general system is tractable if we use the wrong objective function. Alternatively, a simplified system (*i.e.*, assuming $\partial P_i/\partial E_j = 0$ for $i \neq j$) is tractable using a realistic objective function. However, the general system using the realistic objective function is challenging. Our way forward is to re-structure the problem to isolate types of attack affected by different types of effort. By including the major attack vectors, the model is necessarily more complicated than that yielding (5), but will allow insight on how to manage a portfolio when minimizing $L+E$.

5.1 Attack Classes and Attack Vectors

We partition attacks into three classes:

- **Class I attacks (*FULL*)**: these compromise all password-protected accounts of a user. They involve general attack vectors targeting the client machine. Upon success, the attacker acquires actual passwords. Example: client-side malware (e.g., persistent keyloggers), which we assume provides attacker access to all of a user's passwords.

- **Class II attacks (*GROUP*):** these compromise all of a user's accounts protected by the same shared ("group") password, with the attacker obtaining that password; this includes singleton groups. Examples: phishing, brute-force and other guessing, shoulder-surfing, server break-ins to obtain password files, network channel compromise. We assume the attacker will try appropriate credentials with this password on all relevant sites (a finite number), determining associated account userids from public information or otherwise, and gain access to all accounts that use this password. Com-

	Class I (Full, direct)	Class II (Group, direct)	Class III (Single, indirect)
Attack Vectors	Client-side malware (keyloggers, *etc.*)	Phishing, password guessing, shoulder-surfing, system-side database compromise, network channel compromise	Session hijacking, cross-site scripting, password reset mechanisms
Effort elements addressing attack	Run AV, disable unused apps/interfaces, run up-to-date software (apply patches), avoid suspicious links, don't click on email attachments	Choose strong passwords, don't re-use passwords, change passwords often, don't write down, avoid phishing sites	*little advice*

Table 1: Attack classes for password-protected accounts, attack vectors, and relevant user effort elements (defensive actions).

promising one account thus may imply losses in all same-password accounts of the user.

- **Class III attacks (*SINGLE*):** these compromise only a target account, without obtaining the actual password.[4] Example attack vectors: cookie stealing, single-session hijacking (e.g., by cross-site request forgery), exploiting password reset vectors (but not those that mail-back original passwords). The attacker may gain account access, but cannot leverage this to access other accounts, even same-password accounts.

While this classification is still a simplification—e.g., some passwords are easily derived from related passwords [49]—it allows us to model cross-contamination. To handle the case where passwords are modified-and-shared rather than simply shared between groups, as observed by Das etal [17], would require an adjustment to this model (e.g. by modifying Class II). Table 1 synopsizes the attack classes, their principal vectors and user effort that addresses them. Note that the user effort related to passwords (*e.g.*, strength, avoiding re-use, avoiding phishing sites) is concentrated in Class II. Class I deals with system-wide attacks. Class III deals with attacks affecting only a single account, not others sharing the same password.

The probability of individual account compromise can now be split as:

$$P_i \approx P^I + P_i^{II} + P_i^{III} \tag{8}$$

where superscripts denote attack class. Here, and throughout the paper, the compromise probabilities are assumed small enough that the well-known approximation $(1-\prod_i(1-P_i)) \approx \sum_i P_i$ can be used. For Class I we omit the subscript from attack probability P^I, since it has the same value for all accounts. Now, if a user has $G \leq N$ unique passwords, sharing password w_j across a set $\mathscr{A}_J$ of accounts, the expected loss becomes:

$$\begin{aligned} L &= P^I \sum_{i=1}^{N} L_i + \sum_{J=1}^{G} \Big(\sum_{i\in\mathscr{A}_J} P_i^{II} \Big) \Big(\sum_{i\in\mathscr{A}_J} L_i \Big) + \sum_{i=1}^{N} P_i^{III} L_i \\ &= P^I \sum_{i=1}^{N} L_i + \sum_{J=1}^{G} P_J \cdot L_J + \sum_{i=1}^{N} P_i^{III} L_i. \end{aligned} \tag{9}$$

To distinguish, e.g., account i from password-sharing group J, we abuse notation with upper-case indices; and similarly subscripts to denote sums over groups, so

$$L_J = \sum_{i\in\mathscr{A}_J} L_i \quad \text{and} \quad P_J = P_J^{II} = \sum_{i\in\mathscr{A}_J} P_i^{II}, \tag{10}$$

dropping P_J^{II}'s superscript as this is for Class II only.

The three terms on the right side of (9) match the three attack classes. The first term is the probability of a Class I attack, weighted by the entire portfolio value. The second term is the sum across the G password-sharing groups, each weighted by the value of the accounts in that group. This highlights the drawback of password re-use: a compromise is not isolated to one account, but spreads to others. The third term is the sum of probability of individual account compromise weighted by the account value.

5.2 Modeling Effort Allocation and Effectiveness

To minimize an objective function that includes both loss and effort, both must be mapped to the same dimension. For simplicity, we assign a monetary value E for the time and effort—a mapping that is naturally user-dependent. The cost of this management has different components; preventing different attacks often requires different mechanisms. Thus again, this is split based on the class of attack the effort addresses:

$$\begin{aligned} E &= E^I + E^{II} + E^{III} \\ &= E^I + \sum_{J=1}^{G} E_J^{II} + \sum_{i=1}^{N} E_i^{III}. \end{aligned} \tag{11}$$

Under the assumption that effort is applied independently across classes, from (11) we also have: $\partial E/\partial E^I =$

[4] In the case of password resets mentioned next, the attack may recover a new temporary reset password, but not the original password possibly shared across other accounts.

$\partial E/\partial E^{II} = \partial E/\partial E^{III} = 1$. E^I is the cost of defensive effort related to Class I attacks—including, e.g., the total cost and time/effort associated with purchasing/running anti-virus software, and all effort related to keeping a computer malware-free. E^{II}_J is the cost of effort involved in combating Class II attacks on a group that share the same password (brute force, social engineering, etc.). Clearly, $E_G(N)$ given in (2), the cost of managing the password portfolio, is a portion of E^{II}. However, E^{II} also includes effort devoted to other Class II attacks, such as phishing [33]. E^{III} relates to account-specific efforts, which may include, e.g., managing one-time passwords or second-factor authentication devices.

Assuming the three types of efforts can be controlled independently, objective $L+E$ is minimized when

$$\frac{\partial(L+E)}{\partial E^I} = \frac{\partial(L+E)}{\partial E^{II}} = \frac{\partial(L+E)}{\partial E^{III}} = 0 \qquad (12)$$

which simplifies to:

$$\frac{\partial L}{\partial E^I} = \frac{\partial L}{\partial E^{II}} = \frac{\partial L}{\partial E^{III}} = -1. \qquad (13)$$

Substituting our expression for loss (9) into each of these three equalities, the parade of equations concludes with:

$$\left(\sum_{i=1}^{N} L_i\right) \frac{\partial P^I}{\partial E^I} = -1 \qquad (14)$$

$$L_J \cdot \frac{\partial P_J}{\partial E_J} = -1, J = 1 \cdots G \qquad (15)$$

$$L_i \cdot \frac{\partial P^{III}_i}{\partial E^{III}_i} = -1, i = 1 \cdots N. \qquad (16)$$

Note that we have used the fact that the effort devoted to group J does not affect either the probability of loss for group K (*i.e.*, $\partial P^{II}_K/\partial E^{II}_J = 0$ when $K \neq J$) or the effort devoted there (*i.e.*, $\partial E^{II}_K/\partial E^{II}_J = 0$ for $K \neq J$).

5.3 Implications of the Model

Equations (14)-(16) help formalize the concept of optimization of defensive investment (i.e., effort) related to expected loss. We briefly discuss each further.

Class I equation. Eqn (14) relates to Class I attacks, e.g., client-end malware like keyloggers. It isolates the cost of avoiding such attacks from efforts directly related to password management. The sum over all L_i reflects the definition: Class I attacks compromise *all* of a user's passwords—thus the loss may be quite large, especially if the sum strongly dominates individual L_i values. The absence of individual P_i in (14) reflects that defensive effort (cost) related to reducing likelihood of Class I losses is unrelated to costs associated with managing individual passwords. This is notable as current password advice to end-users *is* predominantly related to managing individual passwords (e.g., choosing stronger, more complex passwords, not re-using across accounts), none of which is related to (14).

Common advice related to (14) includes (see Table 1): keeping software up-to-date with patches; using AV (anti-virus) protection; disabling unused applications and interfaces; "hardening" the platform OS.

Regarding overall investment in client-end protection, (14) informs us that effort expended defending Class I attacks should be driven by: (i) the total value of all accounts the user accesses from the client device—the larger this value, the more worthwhile even small defensive efforts which reduce the probability of losses; and (ii) the degree to which incremental defensive effort reduces the probability of Class I attacks. Note that, counter-intuitively, the effort optimally expended is *not* driven by the absolute probability of Class I attacks—since effort spent doesn't necessarily reduce the probability of successful attack, even if P_i is large.

Class II equation. Note that (15) is a set of equations, one for each password-sharing group J. L_J accumulates losses over the accounts sharing a password, based on the assumption that once a password is compromised, all accounts sharing it may suffer. P_J sum probabilities over all accounts in the group, for a similar reason.

Regarding overall investment in defenses against Class II attacks, (15) informs us that the allocation of such effort should be driven by the following, considered now *for each group:* (i) the total value of all accounts in the shared-password group—the larger this value, the more worthwhile defensive efforts which reduce the P_J; and (ii) the cumulative sum, across all groups accounts, of the degree to which incremental defensive effort reduces the probability of Class II attacks. As above for Class I, the optimal effort expended is *not* driven by the absolute probability of Class II attacks; the same is true for (16) and Class III.

The similarity between (15) and (5) should be obvious: we again have a constraint involving a single partial derivative. A few conclusions can be drawn that mirror those drawn about the simpler model in Section 4. First, all passwords should *not* be equally strong (that would be wasteful, allocating excessive effort to low-value account groups at the expense of high-value ones). Second, the rate of change of P_J with respect to effort should be inversely proportional to L_J. This means that (unless a user has excess capacity of effort they wish to spend, and no higher-value groups to spend it on) groups with $L_J \approx 0$ *should* be very exposed and *should* have weak passwords, since as $1/L_J \to \infty$, they should be at the point where $\partial P_J/\partial E_J$ is extremely high; thus even tiny invested effort would reduce P_J significantly, but spending effort there would be wasteful as we care not about

P_J but $P_J \cdot L_J$. Effort is better spent on an account group with high L_J (even if $\partial P_J/\partial E_J$ is very low). It makes no sense to invest at all on accounts where $L_J = 0$, so long as any other account has $L_J > 0$.

Toy example. To illustrate (15), suppose two bank accounts sharing a common password have loss values 10 and 12. Assume that the first account is phished, and thereafter an attacker tries the same password with appropriate obtained userid on all banks. Assume further that additional effort $\delta E = 3$ units (e.g., a stronger group password) reduces individual account compromise probabilities from 0.1 to 0.09 (first account) and from 0.05 to 0.03 (second). Then the initial expected loss (see (9)) of $(10+12)(0.1+0.05) = 3.3$ is reduced, by extra effort, to $(10+12)(0.09+0.03) = 2.64$. Thus extra effort of 3 units reduced loss by only 0.66. This can also be observed by looking at the differences (or derivatives, as in (15)); the change is $(10+12)(-0.01/3 - 0.02/3) = -0.22$. And, in this example, as -0.22 is less negative than -1, we have higher investment than optimal—the cost of effort invested exceeds the reduction in loss it provides. The equations thus confirm our expectations, despite the "units of measure" carrying little meaning.

Class III equation. Finally, (16) reminds us that, regardless of password policies, we must keep in mind and beware reset mechanisms and alternative access paths. Class III attacks involve only a single account and are unrelated to group sharing of passwords, being unrelated to the actual choice of passwords. As noted in Table 1, users get little advice related to Class III attacks (and hence $\partial P_i^{III}/\partial E_i^{III} \approx 0$). In the sequel, (16) is discussed little, as risks associated with these attacks are largely impervious to user effort, our present focus. Regarding overall effort defending Class III attacks, (16) tells us that, considering now *each account individually,* the allocation of such effort should depend on: (i) the account value; and (ii) the degree to which new effort reduces the probability of Class III attacks on it.

6 Account Grouping for Password Re-use

We saw in Section 3 that, without additional coping mechanisms, re-use is unavoidable for large N. We now show that it can help, even for smaller portfolios. Since we seek to minimize L+E there are two components to consider: changes in effort, and in expected loss. For loss, we need consider only Class II attacks, as Class I and III attacks are unaffected by re-use.

Consider the case of three accounts, two relatively low-value (L_1, L_2), one high-value (L_3) so $L_3/(L_1 + L_2) = m >> 1$. For simplicity assume further $P_1^{II} \approx P_2^{II}$ (we will drop superscripts II, as only Class II attacks are relevant). Now compare Case A (using three unique passwords) vs. Case B (re-use one password across low-value accounts, with unique password for high-value). For Case A, expected Class II losses are: $P_1L_1 + P_2L_2 + P_3L_3$. For Case B, re-use increases the expected loss over the first two accounts by $\Delta L = (P_1+P_2)(L_1+L_2) - (P_1L_1 + P_2L_2)$; as $P_1 = P_2$ now, this is $P_1L_2 + P_2L_1 = P_1(L_1+L_2) > 0$, but the user manages one fewer password. Assume the saved effort ΔE is used to strengthen the high-value password[5] reducing the expected loss related to the third account from P_3L_3 to $(P_3(1-e))L_3$ where $0 < e < 1$. So Case B is preferable (has lower expected loss) provided the increase ΔL in expected loss over the first two accounts is less than the expected decrease on the third, i.e., provided: $P_1(L_1+L_2) < eP_3L_3$, or equivalently,

$$m > P_1/(eP_3) \tag{17}$$

We expect (17) often holds—e.g., if $m = 50$ (a financial account with value 100 times that of a free or low-value subscription site) and $P_1 \approx P_3$, then (17) is true for $e > 1/50 = .02$, i.e., a 2% or greater reduction in probability of loss due to a strengthened password. The right side of (17) becomes even smaller if $P_3 > P_1$, and if $P_3 < P_1$ then (17) still holds for a correspondingly larger e. Thus certainly, re-use can be beneficial.

Of course, guessing is but one possible Class II attack; some others also increase the consequences of re-use. The risks of some, like phishing, can be reduced by the user, while that of others, like server-side attacks, are largely impervious to user effort (see 7.3).

6.1 Share among Accounts of Similar P/L

We now explore how to re-use passwords "properly". Based on the loss model, we give an optimal password re-use strategy in the following sense: for a fixed number of passwords, and a given set of accounts (thus effort is fixed), find how to group accounts to minimize total expected loss.

As before, assume a user splits N accounts into G groups each sharing a unique password. Per the second term on the right of (9), the total Class II loss is:

$$L^{II} = \sum_{J=1}^{G} (\sum_{i \in \mathscr{A}_J} P_i^{II})(\sum_{i \in \mathscr{A}_J} L_i) \tag{18}$$

Is there an optimal way to partition this set of accounts into shared-password groups $\mathscr{A}_J$?

We first address the case of adding a new account to an existing portfolio; i.e., we have G groups and must decide to which group a new account is best added. From (18), adding a new account with (P_i, L_i) to group J, the incremental loss is (with L_J, P_J as in 5.1):

$$\Delta L = P_iL_J + L_iP_J + P_iL_i \tag{19}$$

[5] If users do not do this, the case for re-use is lost; this is critical.

From (2), the incremental effort is $\Delta E \approx \lg G$. Since neither ΔE, nor the third term of ΔL depend on the group J, the objective function, $L+E$, depends on the group assignment only through the first two terms of (19). Thus the new account should be added to the group $\mathscr{A}_J$ minimizing $P_i L_J + L_i P_J$. This brings an interesting insight: if any group J exists such that $P_J < P_K$ and $L_J < L_K$ for all G (i.e., the group has both a smaller total probability and a smaller total loss than all other groups), then all new accounts should be added to that group J, until one of the two inequalities fails.

Thus without loss of generality, the remaining case is in deciding between two groups $\mathscr{A}_J$, $\mathscr{A}_K$ when $P_J < P_K$ and $L_J > L_K$. Here, new account i should be assigned to $\mathscr{A}_J$ (vs. $\mathscr{A}_K$) if and only if:

$$P_i L_J + P_J L_i \leq P_i L_K + P_K L_i \tag{20}$$

This can be rewritten as

$$\frac{L_i}{P_i} \geq \frac{L_J - L_K}{P_K - P_J} \tag{21}$$

Fig.2 illustrates this constraint graphically. Recall that a line of slope m in the PL plane is given by $L = m \cdot P + c$. Thus, (21) says that account i should be placed in group $\mathscr{A}_J$ (vs. $\mathscr{A}_K$) if and only if point (P_i, L_i) lies above a line with slope $(L_J - L_K)/(P_K - P_J)$ going through the origin. Fig.2 shows the construction of a (solid red) line with slope $(L_J - L_K)/(P_K - P_J)$; it passes through points (P_K, L_J), (P_J, L_K). The dashed red line is one of the same slope, but through the origin.

In summary, the decision boundary between adjacent groups $\mathscr{A}_J$ and $\mathscr{A}_K$ is given by the line:

$$L = \left(\frac{L_J - L_K}{P_K - P_J} \right) \cdot P. \tag{22}$$

A necessary condition for optimality is the absence of *profitable single moves* in the following sense: if a partitioning of accounts is optimal, the total loss cannot be decreased by moving any account i from group $\mathscr{A}_J$ to any other group $\mathscr{A}_K$. This can be expressed as

$$P_i L_{J^*} + P_{J^*} L_i \quad \leq \quad P_i L_K + P_K L_i \quad \text{for all } K. \tag{23}$$

Here (P_K, P_{J^*}) are the total loss probabilities, and (L_K, L_{J^*}) the total losses, resp., for groups K and J^* where J^* denotes group J after removing account i. Similar to (21), we can rewrite (23) as

$$\frac{L_i}{P_i} \geq \frac{L_{J^*} - L_K}{P_K - P_{J^*}} \quad \text{for all } K. \tag{24}$$

Consider the case when the number of accounts N becomes large, in which case P_i and L_i are typically small relative to P and L. We can then assume the total loss and probability of each group does not change much by adding or removing a single account. Thus $P_{J^*} \approx P_J$ (i.e., $P_J \approx (P_J + P_i)$).

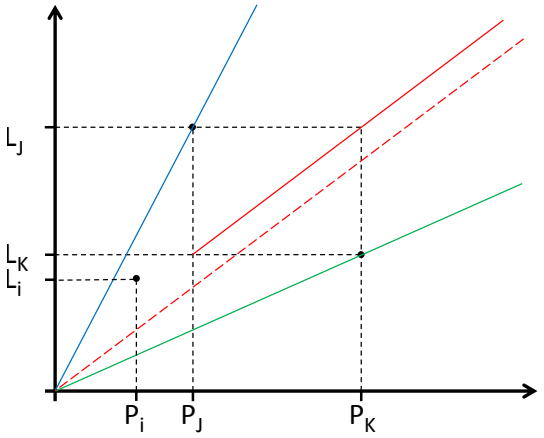


Figure 2: Optimal assignment of a new account (P_i, L_i) between two groups J and K. If the new account falls above the dashed red line, total loss will be smaller when the account is assigned to group J.

We first show that given an optimal grouping, for any groups J and K the decision boundary is bounded by:

$$\frac{L_K}{P_K} \leq \frac{L_J - L_K}{P_K - P_J} \leq \frac{L_J}{P_J}. \tag{25}$$

The decision boundary slope (Fig.2, dashed red line) thus must be between that of the green and blue lines.

To show this, note that group K must contain at least one account i with $L_i/P_i \geq L_K/P_K$ (since all of the L_i and P_i are ≥ 0). Thus (21) holds, implying account i belongs in group $\mathscr{A}_J$ rather than $\mathscr{A}_K$ unless the righthand inequality of (25) holds. The reverse argument applies to show the lefthand inequality in (25).

Now (22) tells us that the decision boundaries are lines through the origin; so each group has at most two neighbors. Further, (25) when applied to every pair of "adjacent" groups in the PL plane, implies the same ordering applies to not only the ratio of L and P differences as in (22), but also the ratio of their values:

$$\frac{L_1}{P_1} \geq \frac{L_2}{P_2} \geq \cdots \geq \frac{L_G}{P_G} \tag{26}$$

where, without loss of generality, the groups have been ordered clockwise, according to their order in the PL plane. In general for groups $\mathscr{A}_J$, $\mathscr{A}_K$, recall that $P_J < P_K$ implies $L_J > L_K$. From this it follows that, given an ordering for the ratio, the same ordering must apply to the expected loss and the reverse ordering for probability, i.e.,

$$P_1 \leq \cdots \leq P_G \quad \text{and} \quad L_1 \geq \cdots \geq L_G. \tag{27}$$

Thus ordering the account groups by decreasing total loss, they have increasing total probability; due to the possibility of equality, none of the orderings is strict.

6.2 Groups Similarly Weighted by PL

Consider next how large and how disparate different groups will be. We show that under certain conditions, the groups formed have similar individual products PL. With focus again on the outcome as G increases, from Section 6.1 the groups obey an ordering in terms of P, L, and L/P, and the decision line slope (dashed red line in Fig.2) must be between the slopes of the two adjacent groups. Thus, assuming accounts exist around every point in the PL plane, as G increases the adjacent groups have increasingly similar slopes, with bounds on the decision boundary slope per (25). Since, from (27), the L_i are non-increasing, and from (27), the P_i are non-decreasing, we have $L_J \geq (L_J + L_K)/2 \geq L_K$ and $P_J \leq (P_J + P_K)/2 \leq P_K$. It follows from (25) that, as G increases:

$$\frac{L_J - L_K}{P_K - P_J} \approx \frac{(L_J + L_K)/2}{(P_J + P_K)/2}. \tag{28}$$

Re-arranging yields:

$$(L_J - L_K)(P_J + P_K) \approx -(P_J - P_K)(L_J + L_K) \tag{29}$$

Expanding products and eliminating common terms,

$$P_J L_J \approx P_K L_K \tag{30}$$

Thus the product of probability and loss for adjacent groups is about equal, increasingly so as the numbers of groups G and accounts per group increase.

6.3 Pedagogical Illustration through Two Generated Datasets

To illustrate, we generated two datasets, assigning accounts to groups with an optimization program obeying the "no profitable moves" rule. The simulation models 100 accounts, with randomly assigned P_i and L_i, to be divided in five groups (shown by different colors in the figures). The program assigns accounts to a group one at a time. After each assignment, it tests all possible single moves and swaps, performing any profitable moves before moving on to assign the next account. In the first dataset, corresponding to Fig.3 and Table 2, P_i and L_i are independently drawn from uniform distributions.

In practice, the combination of (23), (25), and (30) means that whenever passwords are to be re-used across accounts, the optimum strategy is to do so across accounts with similar P/L ratio, and add enough accounts per group to achieve similar total PL products for each group. The resulting account assignments split the PL plane into slices (see Fig.3). This implies that most high-value accounts end up in the same group (particularly if they have low compromise probability), and most low-value accounts end up in another group (particularly if they have high compromise probability)—apparently in line with what many users currently do. Table 2 reports selected characteristics of the 5 password groups; note the similar values of PL across groups, strictly decreasing L, and strictly increasing P and P/L.

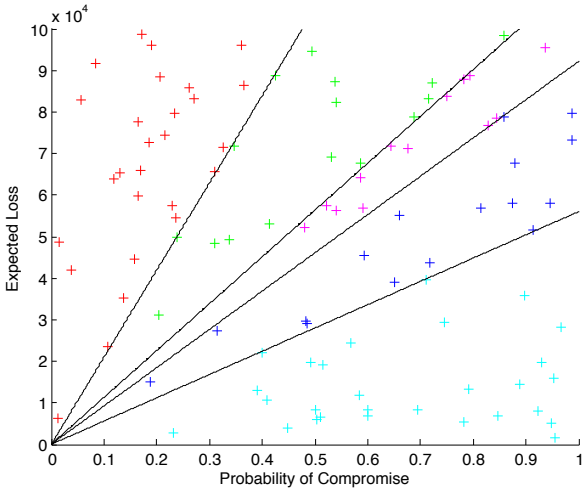


Figure 3: Password grouping, under the "no profitable moves" strategy. For this example, 100 accounts are uniformly placed at random in the PL plane, and optimally assigned to one of 5 groups. Note the linear decision boundaries, corresponding to P/L ranges (slices).

While the dataset used to produce Fig.3 allows visualization of the linear decision boundaries, such a dataset with independent distribution over P and L is not what we would expect in practice. We thus generated a second dataset (see Fig.4 and Table 3) where L_i follows a power law distribution and the expected value of P_i is inversely proportional to (the square of) L_i. While all observations on the previous dataset still hold, further insights are evident. As on this dataset high-value accounts are less likely to have high P_i, the high-value accounts end up grouped together. Indeed, group 1 includes 53 accounts, more than half of the set, while group 5 has only 4 accounts (see Table 3).

The total resulting loss across all five groups is 7.94×10^4. To see how this optimal assignment compares to a random assignment, we computed total loss on the same dataset on randomly assigning accounts to the 5 groups (in 100,000 Monte Carlo trials), finding an average PL of 1.16×10^8 (std deviation 0.24×10^8). Thus the optimal loss was 1500 times smaller than by random assignment, and 5 standard deviations below the mean.

We emphasize that both datasets are modelled examples to illustrate principles. For other datasets, the general findings will hold, but actual construction of groups may significantly differ depending on the data.

Group #	P	L	PL	P/L	max P/L	min P/L	Group Size
1	1.88e+01	3.96e+05	7.44e+06	4.75e-05	6.09e-04	1.80e-05	28
2	1.14e+01	8.08e+05	9.17e+06	1.40e-05	1.77e-05	1.09e-05	16
3	9.35e+00	9.71e+05	9.08e+06	9.63e-06	1.08e-05	8.93e-06	13
4	7.94e+00	1.14e+06	9.06e+06	6.96e-06	8.72e-06	4.79e-06	16
5	4.91e+00	1.82e+06	8.93e+06	2.70e-06	4.73e-06	3.22e-07	27

Table 2: Characteristics of each group in the grouping corresponding to Figure 3.

Group #	P	L	PL	P/L	max P/L	min P/L	Group Size
1	2.29e+01	3.24e+02	7.41e+03	7.06e-02	9.46e+02	1.44e-03	53
2	6.70e-01	1.76e+04	1.18e+04	3.80e-05	1.24e-03	4.45e-06	24
3	6.42e-02	2.40e+05	1.54e+04	2.68e-07	1.66e-06	3.01e-08	11
4	7.04e-03	2.45e+06	1.72e+04	2.88e-09	1.66e-08	4.09e-10	8
5	1.26e-03	2.19e+07	2.77e+04	5.76e-11	2.80e-10	9.29e-12	4

Table 3: Characteristics of each group in the grouping corresponding to Figure 4.

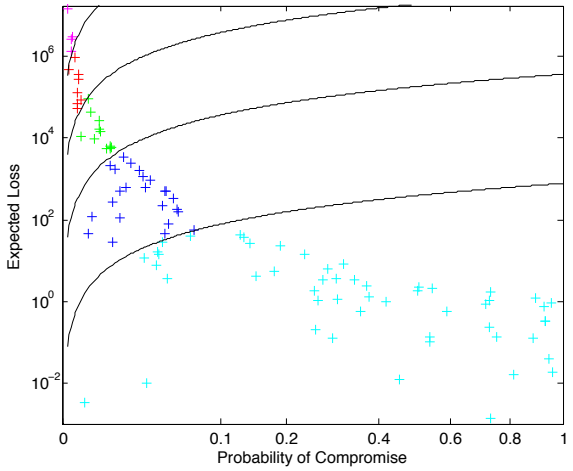


Figure 4: Password grouping, second dataset (P_i's drawn from a distribution with mean inversely proportional to L_i^2). Due to the non-linear axis, the decision boundaries are no longer linear. The number of accounts in each group differs from Fig.3.

7 Special Cases

Next, special cases illustrate how the model addresses additional assumptions and circumstances.

7.1 Case 1: Unknown P_i (Modeled as Equal)

The model highlights two variables with large effect on the problem: loss and compromise probability. Most users could give some estimate of loss that would result from compromise of a specified account—perhaps not entirely accurate, but representative of expected loss, even if only in relative terms. In contrast, user estimates of probabilities would likely be far worse, perhaps without sense of even relative P_i's. We thus consider here what results from the optimization model on assuming equal probabilities $p = P_i$ for all i. Slice-based partitioning still applies, as does the ordering—the latter now easier with all accounts on a vertical line in the P_iL_i plane. The main question is how many accounts will each group have, and how does that relate to the L_i of accounts in each group.

If group J has N_J accounts, write $P_J = pN_J$. Then (30) yields $(pN_J)(L_J) \approx (pN_K)(L_K)$, or, equivalently:

$$\frac{N_J}{N_K} \approx \sqrt{\frac{L_K/N_K}{L_J/N_J}}. \tag{31}$$

Thus groups with high-value accounts will have fewer accounts; optimally, the number of accounts N_J in a group J varies inversely with the square root of the average loss L_J/N_J in that group.

To illustrate, we re-run the optimization process on the second dataset (see Section 6.3), but now assuming ignorance of individual probabilities, modeling equal P_i. The principles discussed earlier now result in the accounts being split by strict ordering of losses. The number of accounts in the 5 groups is now (82, 11 4, 2, 1), vs. (53, 24, 11, 8, 4) in Table 3. As might be expected, total losses increase to 1.24×10^6, vs. 7.94×10^4 for optimization using known probabilities. This is 16× higher than the optimum, but still 93× smaller than the average loss from random assignment (see Section 6.3).

7.2 Case 2: Group Passwords of Unequal Strength

We showed in Section 5.3 that passwords should *not* have the same strength. We now show how the assumption made in Section 6 (that P_i did not change much when we moved account i from password group J to group K) can be relaxed, so that there is no incompatibility. Here we

briefly analyze the impact, on optimization results, when P_i is password-dependent and groups have passwords of different strength. Denote the (now group-dependent) compromise probabilities $P_{i\in J}$, $P_{i\in K}$. Then by the argument used in (20), account i should be assigned to $\mathscr{A}_J$ if and only if

$$P_{i\in J}L_J + P_J L_i \leq P_{i\in K}L_K + P_K L_i. \tag{32}$$

We again seek a bounding condition on L_i/P_i, but now using what group-neutral P_i value? We use the geometric average $P_i = \sqrt{P_{i\in J}P_{i\in K}}$ and define the squareroot ratio $r = \sqrt{P_{i\in J}/P_{i\in K}}$. Then (32) yields

$$\frac{L_i}{P_i} \geq \frac{rL_J - (1/r)L_K}{P_K - P_J}. \tag{33}$$

Note $r>1$ if group J has password weaker than K. Thus with respect to group assignment, a weaker group J password has an effect equivalent to scaling up group losses L_J, making it harder to satisfy the condition for assignment to group J. Other results regarding the slicing, P and L ordering, and so on remain as before.

7.3 Case 3: Unequal Server Break-in Probabilities

Finally, consider the effects of different levels of security at the server. The probability of server break-in is largely outside users' control, but the consequences are not: a user may decide to share a password across accounts, only to have one of the servers leak her password, compromising all accounts sharing it. While the previous analysis already takes into consideration server break-in (as a Class II attack), we now analyze how two sites with different server break-in probabilities will affect the optimum allocation.

Consider two accounts i and j, with same values $L_i = L_j$ but different probabilities, $P_i = P_j + \delta_i$, where δ_i is the added break-in probability due to a site i server poorly managed compared to j. Upon assigning account i (poorly managed) to a group, the added probability δ_i will imply a higher ratio P_i/L_i, so the account will (likely) be grouped with accounts with higher P/L, typically lower-value accounts. Furthermore, as discussed in Section 5.3, these groups may have a weaker password. Thus, for a server with higher break-in probability, optimum password grouping seems to push towards grouping the related account with lower-value accounts.

Related to this, our criteria for optimality depend on how loss probabilities change with respect to effort, but not on the magnitudes of the probabilities themselves. Consider the possible case of a threat unaddressable by user effort, swamping all others. Let $P_i = P_{i,u} + P_{i,\bar{u}}$, where $dP_{i,\bar{u}}/dE = 0$. If $P_{i,\bar{u}} > 10^3 P_{i,u}$, it may be fruitless to spend substantial user effort if such expenditure affects only the third decimal place in P_i. Nonetheless, this is what our criterion for optimality suggests. System-side or back-end (server) risks may swamp risks under user control; we simply do not know.

7.4 Case 4: Coping Alternatives including Password Managers

Despite violating long-standing password guidance, writing passwords down is, if properly done, increasingly accepted as a coping mechanism. Other strategies to cope with the human impossibility of using strong passwords everywhere without re-use include single-sign-on, use of email-based password reset mechanisms, and password managers. Such "password concentrators", a form of password re-use, allow access to many accounts from one master access point, with account passwords stored either locally or in the cloud. While not explored in detail here, each can be analyzed in our framework; we illustrate for password managers.

The main threats (recall Table 1) when re-use is employed are client-side malware (all accounts fall), and various Class II attacks such as guessing, phishing, sniffing wireless links and server breaches (all accounts in the same sharing group fall). We must modify this picture slightly if a password manager is used. For Case A (password store on a user's local machine), the main risk is still Class I attacks like client-side malware. There is a decreased risk of phishing presumably, as users remember fewer individual passwords; similarly for guessing attacks, as arbitrarily strong passwords now require no user effort, and the master password that unlocks the store resides on the client. A server-side breach compromises only a single account. Thus, a password manager with client-side store approximates our model with $G = N$. The cost, of course, is that portability across different client devices is lost as the passwords (if they are unique and random) are effectively anchored to the client on which they are stored.

Consider next (Case B) a cloud-based store, protected by a single password. Phishing and guessing attacks against any system-assigned secrets at the end-servers remain unchanged. Now however, additional guessing, phishing and server breach attacks exist against the single master password which can result in the compromise of all accounts. Class I attacks (e.g. due to malware on the client) are unchanged. A password manager with a password-protected cloud-based store approximates our system with $G = 1$. It trades one set of risks for another: the use of random and unique passwords in such a system reduces both the risks related to any single manager-chosen password being stolen and those related to re-use in the face of server compromise. However, it introduces

severe new risks: if the master password is guessed or used on any malware-infected client, or the cloud store is compromised, then all credentials are lost.

8 Discussion and Implications

Recapping, recall first the task of end-users: to choose passwords random and strong (entropy $\lg S$ bits) without re-use. The effort to manage N such passwords without re-use is modelled as $N \lg S + \lg(N!)$; as portfolio size increases, this overwhelms user capability.

M1: *Remembering random and unique passwords is infeasible for other than very small portfolios.*

Users coping strategies include weak passwords and re-use. There is a large disconnect: what standard advice mandates as essential turns out to be impossible. We suggest this is due to a failure to explicitly include user effort in the objective function. Seeking to minimize loss alone leads to unrealistic effort-maximizing solutions. While some recent work [5, 4, 29] criticizes the practice of ignoring the burden password advice places on users, it has not to our knowledge been included directly in the objective function. We make a related observation:

M2: *While advice typically minimizes L over a single or small set of sites, user best interest is to minimize $L+E$ over an entire portfolio.*

The diversity of attacks complicates our search for an optimum effort allocation. Short-cuts are tempting; we can minimize $L+E$ while ignoring cross-account attacks (as in Section 4), or consider all attack types and minimize L alone. The first scopes the problem too narrowly, the second leads to the unrealistic demand to invest unbounded effort. While both yield "solutions" that are simpler than the model in Section 5, our work suggests that realistic analysis must address a realistic attack model *and* a realistic objective function.

M3: *Realistic analysis of password effort allocation requires incorporating attack vectors affecting 1) all accounts; 2) accounts sharing a password; and 3) single accounts.*

Our segmentation of the space into Class I, II and III attacks yields interesting insights. Minimizing $L+E$ over a portfolio implies user effort be spent unequally across accounts. As can be seen from (15), all passwords should not be equally strong; equal spending overspends on low-value, and underspends on high-value accounts (or account groups). Recall that, from (27), there is an ordering of the group values L_J; the largest may be many times greater than the smallest ($L_1 \gg L_G$). Any group for which $L_J \approx 0$ should have $\partial P_J/\partial E_J$ high (meaning a weak password). If we again invoke the reciprocal relation between P_J and E_J suggested in Section 4, we'd again find $E_1 = \sqrt{L_1/L_G} \cdot E_G$. Thus a $10^4\times$ value difference between the most and least valuable groups would imply a $100\times$ difference in invested effort. In this sense, not only are weak passwords understandable and allowable, but *their absence* would be sub-optimal:

M4: *A password portfolio strategy that rules out weak passwords is sub-optimal.*

Next, while sharing a password across a group of accounts can amplify consequences if it is compromised, we find it is sub-optimal *not* to re-use. First, (1) indicates re-use becomes unavoidable when N is large. Second, (2) and Fig.1 demonstrate the tradeoff involved even if N is small enough that re-use is theoretically avoidable; i.e., re-use increases the probability of loss from certain attacks, but also reduces effort. The question then is not whether re-use is good or bad, but whether the effort required to avoid re-use can be better spent on other attack types. Section 6 gives an example.

M5: *A password portfolio strategy that rules out password re-use is sub-optimal.*

The optimal strategy places accounts with similar P/L ratio in groups sharing a password. Enough accounts are added to each group to achieve similar PL products per group. Most high-value accounts (particularly if they have low P_i) end up in the same group(s), and most low-value accounts (particularly if they have high P_i) in another group(s).

M6: *Optimal password grouping tends to (i) group together accounts with high value and low probability of compromise; and (ii) group together accounts of low value and high compromise probability.*

The above observation lines up well with anecdotal accounts of what many users actually do. Our findings also agree with the informal claim [29], that users' actual effort allocation represents an efficient operating point. Thus, actual user password-related behavior is closer to optimal than current expert advice.

Password managers (cf. Section 7.4) may improve usability and reduce some risks, but remain vulnerable to Class I attacks (e.g., client-side malware). Managers that store passwords only on the client improve resistance to Class II attacks, since they can choose better passwords and eliminate re-use. However, in storing only on the client this gives up one of the major advantages of passwords, i.e. portability. Managers that store passwords in the cloud remove this restriction, but introduce a new system-wide attack: as before if the client is infected with malware all accounts are compromised, but now this

happens also if the cloud store is breached or the master password is stolen or guessed. Thus, cloud-storage managers trade one type of vulnerability for another.

M7: *Password managers using client-only storage allow a portfolio with random passwords and no re-use, but lose cross-client portability. However, if cloud storage is used it resembles a portfolio with only one group, since a new attack on either the master password or the store itself threatens all accounts.*

Another disconnect stems from many password-related threats being unrelated to the standard advice on maintaining a portfolio: Class I attacks, server breaches and Class III attacks are not reduced by password advice staples such as A1 and A2. Since successful Class I attacks sum the losses across all accounts, the advice to protect against them is disappointingly vague, while advice to protect against the less consequential Class II attacks is far more detailed and effort-consuming. It appears that users are given the advice that is most easily given, rather than the advice that would have greatest impact. Comparing (14) and (15) shows that at optimality the marginal return on effort spent on Class I attacks should be lower than that for any Class II group (*e.g.*, effort should not be wasted strenghtening passwords for a group with low L_J if any effective Class I measure remains undone). Greater focus is needed to explore which advice, for example from Table 1, provides protection against which attack vectors:

M8: *We lack metrics for the cost to end-users, of following standard advice, and the effectiveness of following it on reducing overall expected loss.*

An important outcome of our review is that, when minimizing $L+E$, optimality depends on the losses L_i, and on how the probability of loss varies with respect to effort $\partial P_i/\partial E_i$. In contrast if one minimizes L, the solution depends on neither. Without better knowledge of real-world values for L, and especially $\partial P_i/\partial E_i$, we are unlikely to achieve optimal resource allocation in practice. Conventional user behavior appears to be based almost exclusively on L, which users may be able to estimate; $\partial P_i/\partial E_i$ values are almost entirely overlooked. This points to an important research direction: while recent work has greatly improved understanding of password guessing resistance [8], we are almost entirely ignorant on how this evolves with effort.

M9: *Without better estimates of how loss probability changes with effort, we should not expect to be able to allocate effort (even close to) optimally.*

Finally, can concrete advice for users be distilled from our findings? For example, absent knowing how P_i change as a function of various types of effort, we lack a prescriptive way to determine the optimal number of groups G. Nonetheless, the knee of the curves in Fig.1, and what we know of user behavior [24, 21, 14] points to the number of groups being below 10 if no other aids are used. The values of loss probabilities P_i are entirely unknown; expected loss values L_i, or at least relative importance, are more easily estimated or ordered. Thus the variables needed to find an optimal grouping are (and are likely to remain) unavailable to most users. We might however simplify, e.g., assuming all P_i equal, or that P_i values differ by an order of magnitude between heuristically-defined categories (*e.g.*, banks, merchants, throwaway accounts, *etc.*).

While the optimal strategy involves selective re-use and weaker passwords, benefits accrue only if the effort saved is re-deployed elsewhere for better returns. Users must not arbitrarily weaken and re-use passwords. Thus empirical studies are needed to determine if our guidelines can be followed by users.

We hesitate to give definitive advice. First, this requires more insight than our current understanding of L_J and $\partial P_J/\partial E_J$ values allows. Second, we are reminded how far bad assumptions (e.g., minimizing L vs. $L+E$) can lead us astray. Consider, however, a strategy that chooses G in the range 5 to 10, and assigns accounts to groups by value so that the number of accounts in a group is as in Section 7.1. Given the uncertainty about unknown parameters, a strategy like this may be the best we have—and may even be optimal.

9 Concluding Remarks

We have explored the task of managing a portfolio of passwords. A starting point for our analysis was the critical observation that to be realistic, efficient password management should consider a realistic suite of attacks and minimize the sum of expected loss and user effort. Our model yields detailed results; it indicates that any strategy that rules out weak passwords or re-use will be sub-optimal. We have shown that optimality requires forming groups whose accounts in sum have similar PL values ($P=\sum P_i, L=\sum L_i$). This suggests simple guidelines, such as: if P_i is similar across accounts, then optimal grouping will put high-value accounts in smaller (or singleton) groups, and low-value accounts in larger groups. Our findings are consistent with certain user behaviors (e.g., [47]) that contradict accepted advice, offering to justify the behavior and giving evidence for the model's utility. We find that optimally, marginal return on effort is inversely proportional to account values. We note that while password re-use must be part of an optimal portfolio strategy, it is no panacea. Far from optimal outcomes will result if accounts are

grouped arbitrarily.

Acknowledgements. We thank Robert Biddle, Joseph Bonneau, and anonymous referees for their comments which helped improve this paper. The third author acknowledges an NSERC Discovery Grant and Canada Research Chair in Authentication and Computer Security.

References

[1] Stop.Think.Connect. http://www.stopthinkconnect.org/.

[2] US-Cyber Emergency Response Readiness Team: CyberSecurity Tips. http://www.us-cert.gov/cas/tips/.

[3] A. Adams and M. A. Sasse. Users are not the enemy. C.ACM, pages 40–46, December 1999.

[4] A. Beautement and A. Sasse. The economics of user effort in information security. Computer Fraud & Security, pages 8–12, October 2009.

[5] A. Beautement, M. Sasse, and M. Wonham. The Compliance Budget: Managing Security Behaviour in Organisations. In NSPW, 2008.

[6] G. S. Becker. Crime and punishment: An economic approach. In Essays in the Economics of Crime and Punishment, pages 1–54. UMI, 1974.

[7] J. Bonneau. Guessing human-chosen secrets. University of Cambridge. Ph.D. thesis, May 2012.

[8] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In Proc. IEEE Symp. on Security and Privacy, pages 538–552, 2012.

[9] J. Bonneau and S. Preibusch. The password thicket: Technical and market failures in human authentication on the web. In WEIS, 2010.

[10] J. Bonneau and S. Schechter. Towards reliable storage of 56-bit secrets in human memory. In Proc. USENIX Security, 2014.

[11] A. Brown, E. Bracken, S. Zoccoli, and K. Douglas. Generating and remembering passwords. Applied Cognitive Psychology, 18(6):641–651, 2004.

[12] W. Burr, D. F. Dodson, and W. Polk. Electronic Authentication Guideline. In NIST Special Pub 800-63, 2006.

[13] W. Cheswick. Rethinking passwords. USENIX LISA, 2010. http://www.usenix.org/event/lisa10/tech/slides/cheswick.pdf.

[14] W. Cheswick. Rethinking passwords. ACM Queue, 10(12):50–56, 2012.

[15] W. Cheswick, S. Bellovin, and A. Rubin. Firewalls and Internet Security, 2/e. Addison-Wesley, 2003.

[16] S. Chiasson, A. Forget, E. Stobert, P. C. van Oorschot, and R. Biddle. Multiple password interference in text passwords and click-based graphical passwords. In Proc. ACM CCS, 2009.

[17] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The tangled web of password reuse. NDSS, 2014.

[18] R. Dhamija and A. Perrig. Deja vu: a user study using images for authentication. In USENIX Security, 2000.

[19] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley. Does my password go up to eleven? the impact of password meters on password selection. In Proc. CHI, 2013.

[20] D. Florêncio and C. Herley. Where Do All the Attacks Go? Proc. WEIS, 2011, Fairfax, VA.

[21] D. Florêncio and C. Herley. A Large-Scale Study of Web Password Habits. Proc. WWW, 2007.

[22] D. Florêncio and C. Herley. Where Do Security Policies Come From? Proc. SOUPS, 2010.

[23] D. Florêncio, C. Herley, and P. van Oorschot. An Administrator's Guide to Internet Password Research. In Proc. USENIX LISA, 2014.

[24] S. Gaw and E. Felten. Password Management Strategies for Online Accounts. In ACM SOUPS, 2006.

[25] M. Gouda, A. Liu, L. Leung, and M. Alam. Single password, multiple accounts. In ACNS (Industry Track), 2005.

[26] B. Grawemeyer and H. Johnson. Using and managing multiple passwords: A week to a view. Interacting with Computers, 23(3):256–267, 2011.

[27] E. Grosse and M. Upadhyay. Authentication at scale. IEEE Security & Privacy, 11(1):15–22, 2013.

[28] E. Hayashi and J. Hong. A diary study of password usage in daily life. In CHI (note), pages 2627–2630, 2011.

[29] C. Herley. So Long, And No Thanks for the Externalities: Rational Rejection of Security Advice by Users. Proc. NSPW, 2009.

[30] Imperva. Consumer Password Worst Practices. 2010. http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf.

[31] P. Inglesant and M. A. Sasse. The true cost of unusable password policies: Password use in the wild. In CHI, 2010.

[32] B. Ives, K. Walsh, and H. Schneider. The Domino Effect of Password Re-use. C. ACM, 47(4):75–78, 2004.

[33] M. Jakobsson and S. Myers. Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft. Wiley, 2006.

[34] D. Kahneman. Thinking, fast and slow. Macmillan, 2011.

[35] D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. Econometrica: Journal of the Econometric Society, pages 263–291, 1979.

[36] A. Karp. Forum (comment). C. ACM, 47(6):11–12, 2004.

[37] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In Proc. IEEE Symp. on Security and Privacy, 2012.

[38] R. Lemos. Yahoo breach highlights password reuse threat. eWeek. July 7, 2012.

[39] J. Ma, W. Yang, M. Luo, and N. Li. A study of probabilistic password models. Proc. IEEE Symp. on Security and Privacy, 2014.

[40] H. Markowitz. Portfolio selection. The Journal of Finance, 7(1):77–91, 1952.

[41] R. Nithyanand and R. Johnson. The password allocation problem. In WPES, 2013. Nov. 4, 6 pages.

[42] G. Notoatmodjo. Exploring the 'weakest link': A study of personal password security. C.S. Dept., University of Auckland, 2007. M.Sc. thesis.

[43] S. Preibusch and J. Bonneau. The password game: negative externalities from weak password practices. In Decision and Game Theory for Security, pages 192–207. Springer Berlin Heidelberg, 2010.

[44] S. Riley. Password security: what users know and what they actually do. Usability News, 8(1), 2006.

[45] M. Sasse, S. Brostoff, and D. Weirich. Transforming the "weakest link": a human-computer interaction approach to usable and effective security. BT Tech. J., 19(3):122–131, 2001.

[46] A. Shamir. 2002 Turing Award Lecture. http://amturing.acm.org/vp/shamir_2327856.cfm.

[47] E. Stobert and R. Biddle. The password life cycle: user behaviour in managing passwords. In Proc. SOUPS, 2014.

[48] M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In Proc. ACM CCS, 2010.

[49] Y. Zhang, F. Monrose, and M. K. Reiter. The security of modern password expiration: An algorithmic framework and empirical analysis. In Proc. ACM CCS, 2010.

Telepathwords: preventing weak passwords by reading users' minds

Saranga Komanduri, Richard Shay, Lorrie Faith Cranor
Carnegie Mellon University

Cormac Herley, Stuart Schechter
Microsoft Research

Abstract

To discourage the creation of predictable passwords, vulnerable to guessing attacks, we present *Telepathwords*. As a user creates a password, Telepathwords makes real-time predictions for the next character that user will type. While the concept is simple, making accurate predictions requires efficient algorithms to model users' behavior and to employ already-typed characters to predict subsequent ones. We first made the Telepathwords technology available to the public in late 2013 and have since served hundreds of thousands of user sessions.

We ran a human-subjects experiment to compare password policies that use Telepathwords to those that rely on composition rules, comparing participants' passwords using two different password-evaluation algorithms. We found that participants create far fewer weak passwords using the Telepathwords-based policies than policies based only on character composition. Participants using Telepathwords were also more likely to report that the password feedback was helpful.

1 Introduction

Users are often advised or required to choose passwords that comply with certain policies. Passwords must be at least eight characters long. They must contain characters from at least three out of four character categories (uppercase characters, lowercase characters, digits, and symbols). The password should not be based on a dictionary word.

While rules for composing passwords often feel arbitrary and capricious, they respond to a problem of genuine concern: left to their own devices, a significant fraction of users will choose common passwords that attackers may guess quickly. Composition rules were created decades ago under the assumption that minimum-length and character-set requirements would result in passwords that were harder for attackers to guess. It is only in the past few years that researchers have begun to test this hypothesis (and found the evidence to support it far weaker than assumed).

Indeed, password-composition rules feel arbitrary and capricious because, quite simply, they often are. Users can hardly be blamed if they question the credibility of rules that reward those who choose the common password `P@ssw0rd` over those who enter a long randomly generated string restricted to lowercase letters (e.g., to facilitate typing on a touch-screen keyboard) or of password meters that offer irreconcilably different quality estimates for the same string [4]. If we are to prevent users from selecting weak passwords, we must first improve the technology used to identify weak choices, but also overcome any skepticism caused the failure to clearly explain the need for the restrictions being imposed.

Our proposal, *Telepathwords*, is different from previous weak-password prevention schemes in that, as users enter their proposed password, it shows its best predictions for the next character they will type in real time (see Figure 1). Telepathwords makes these predictions using knowledge of common behaviors users exhibit when choosing passwords, common strings they frequently use to construct passwords, and a general model of the user's language. Telepathwords presents users who enter weak passwords with immediate and compelling evidence that their intended password may be easier to guess than they had previously assumed: a display of the characters they are about to type.

We describe the design, implementation, human-subjects testing, public deployment, and user response to the Telepathwords system. The results of our security testing are particularly compelling. In a 2,560-person Mechanical Turk study, passwords created using Telepathwords significantly outperformed (using both entropy and guessing number metrics) those created under length and character composition policies, while remaining as memorable as passwords chosen with the least stringent requirement (an eight-character

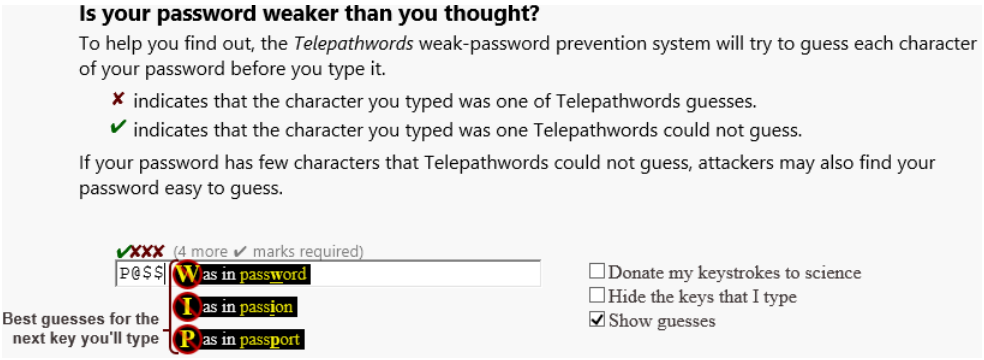


Figure 1: The Telepathwords system, shown here as deployed in a publicly available password-weakness checker, attempts to guess the next character of a users' password *before* he or she types it.

minimum-length requirement). They matched or slightly beat passwords created under a policy that checked against a large cracking-dictionary, while at the same time having more users state that they found the visual feedback useful. The improvements in guessing resistance were most pronounced for the most vulnerable part of the distribution. That is, the weakest passwords created using Telepathwords require orders of magnitude more guesses than the weakest passwords created under policies based on composition and length. This suggests that Telepathwords can offer meaningful improvement in defending against online guessing attacks; an improvement that we hope can rebuild users' confidence that the constraints being imposed on them are indeed necessary.

2 Design and Implementation

We begin our discussion of the Telepathwords system by describing the intended user experience, then discuss the overall architecture and prediction algorithms required to implement that experience. We also describe the feedback mechanisms we included to observe usage of the system, as well as the limitations inherent to our implementation.

2.1 User Experience

Telepathwords enhances the text field into which users type new passwords with two additional elements: a *prediction display* and a *feedback bar*. Figure 1 illustrates both, with the prediction display just to the right of the typed password (`P@$$`) and the feedback bar immediately above it.

2.1.1 Prediction display

The prediction display shows the three characters (or fewer) that Telepathwords predicts the user is most likely to type next. As users are most likely to be familiar with prediction from autocomplete, where the predictions represent a *desirable* mechanism to save labor, we needed to emphasize that the characters telepathwords predict are *undesirable*, as these choices are least likely to make the password harder for attackers to guess. We thus display predicted characters in block uppercase within the prohibition symbol, or 'universal no symbol': a red circle with a slash through it. We anticipated the symbol would be familiar to users because it is standardized (ISO 3864-1, though we did not strive to achieve full compliance in our use), widely used in road signs, and pervasive in popular culture such as t-shirts and movies.

To the right of the character we present a short explanation of why that character was predicted. If we predicted the character because we detected the user typing a repeating sequence of characters, we display 'repeating' followed by the character sequence being repeated. If it is the next character of a common string, we present the words 'as in' followed by that string, with the next character boldfaced and underlined. For example, in Figure 1, the input of "P@$$" yields predictions: "W as in pass**w**ord," "I as in pass**i**on," and "P as in pass**p**ort."

2.1.2 Feedback bar

In the feedback bar above the password-entry field, we show either a checkmark or crossout symbol aligned directly above each of the characters already typed. A checkmark means the character was not predicted by Telepathwords, whereas a crossout indicates it was one of the characters guessed. We also display a crossout if the user types a common substitute for one of the predicted characters, such as an `@` to avoid using an `a`. To the right of these symbols we provide guidance as to how many more hard-to-guess characters are recommended, or would be required if Telepathwords were deployed with a particular minimum hard-to-guess character requirement. For example, Figure 1 shows one check and three crossouts above the user input "P@$$" since each of the last three characters was predicted based on the characters that came before it.

2.1.3 Special cases

In many applications, password-creation fields are configured to hide the keys typed, replacing them with a generic symbol (usually a solid circle or an asterisk). When the password field is configured to hide the characters that have been typed, we also replace those characters with a solid circle in our prediction string. The predicted characters are still shown.

When users type a common substitute for a predicted character, such as a *$* when an *s* is predicted, we display the following message customized for the replacement:

> Replacing a predictable letter with a key that looks similar?
>
> Attackers also know to substitute **$** for **s**, so it does little to improve your password.

We faced a particularly delicate conundrum in how to handle predictions that completed profanities. An examination of the Rockyou leaked dataset reveals that profanities are not uncommon choices. Unlike applications of prediction in search queries, we could not simply remove these predictions, as this would lead users to believe falsely that profane passwords were less weak than they actually are. On the other hand, we could not display profanities to users who might have no intent of typing them, and who might be minors. We decided that providing good security advice mandated that we predict the next character, but we replace the rest of the profane string with a string of solid circles in the explanation of the prediction. We also display a pop-up message if users complete a profanity, alerting them to the fact that profanities are common in passwords and thus quite predictable. In crafting this message, we decided to embrace the inevitability that some users might find humor in our attempts to hide profanity.

> Do you email your mother with that keyboard?
>
> Many people include profanity in their passwords. Attackers know this. If you also use profanity, you'll just make your password easier for attackers to guess.

2.2 Architecture

Telepathwords employs a client-server architecture, using JavaScript to present a front-end user interface using predictions asynchronously queried from a prediction server. The constraints of client-side prediction would not have allowed our prediction engine to use a 1.5GB language corpus (see Section 2.3.1), which we hope to grow in order to increase prediction quality and recognize additional languages.

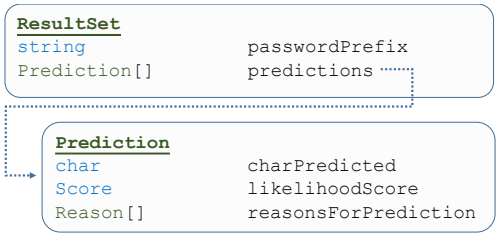


Figure 2: When the client queries the server with a password prefix, the Telepathwords prediction engine generates a result set containing a series of predictions, each of which may have been predicted based on a number of reasons (e.g., a dictionary match or a keyboard pattern).

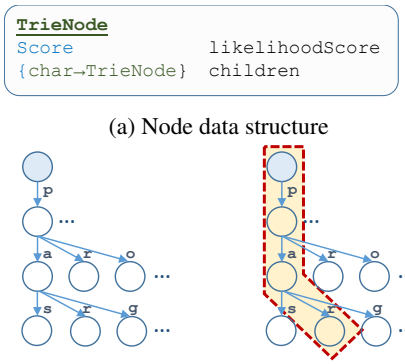
Other weak-password-prevention systems, such as common password meters, eschew server-side predictions. One justification is security. However, the current architecture of the web necessitates that whatever password the user eventually chooses will inevitably be sent to the website's servers in a plaintext-decryptable format. To prevent the size of a prediction from revealing the prefix sent to the server, we use a custom format to compress and then pad responses to a common length. We route all client-server communications over HTTPS.

A second reason to eschew server-side predictions is performance. However, network latencies are relatively small in comparison to users' expectations of response time, and can be made smaller by moving servers closer to users and pre-fetching likely queries, as demonstrated by the speed of auto-complete in web search. For example, though our deployment used servers in a single geographic location to serve users worldwide, the median latency between key-up and the rendering of a prediction at the client was a fifth of a second (see Section 3).

One additional security risk we decided to take was to maintain a cache of previously queried prefixes on the server, whereas we would otherwise be able to delete all evidence of a past request after serving a prediction. This greatly increases the likelihood that when the n^{th} character of a password arrives, the server will already have done the work to process the first $n-1$ characters.

2.3 Prediction algorithms

When performing a prediction, we create a result set data structure and populate it with a set of predictions, as illustrated in Figure 2. Each prediction object represents a possible next character of the password and a score that indicates its estimated likelihood. There may be more than one reason to predict a character, and so each prediction object contains a set of reason objects. We populate the result set by spawning a set of *predictors*, algorithms which identify reasons for predicting a character



(a) Node data structure

(b) A section of the trie (c) Descent to node *par*.

Figure 3: The trie data structure maps strings to likelihood scores. Nodes (circles) with higher scores appear to the left of lower-scoring siblings. Subfigure (c) illustrates a walk to the node storing the likelihood score for string *par*.

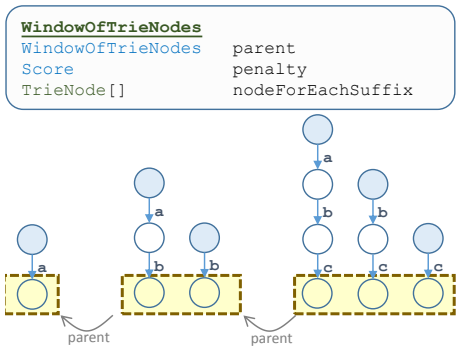


Figure 4: Telepathwords uses a sliding window to walk the trie for each suffix of the queried string. If the query string is a single character (e.g., *a*), the sliding window will contain only one node (left). A two-character string (*ab*) will a sliding window that walks the trie to two different nodes (center). The query string *abc* yields a window that covers the suffixes *c*, *bc*, and the full suffix *abc* (right). Adding one character to the query causes the pointer to each node to descend to the child node for that added character, and creates a new node in the window by stepping from the root node to the added character. Telepathwords may add a penalty to the window when the path down the trie is different from the actual string queried, such as if a window is created to represent a transposition.

will be typed next, add that reason to the prediction object for that character, and increase the predictions score as necessary.

When all the predictors have run, we rank the predictions and reasons. Before sending predictions to the client, we discard predictions and reasons that are not ranked high enough to be displayed to the user. We cache the result set so that we can use it again for future queries for this string, or extensions of this string.

Telepathwords currently contains predictors for common character sequences, keyboard movements, repeated strings, and interleaved strings.

2.3.1 Common character sequences

This predictor detects known prefixes of common character sequences from language models and databases of common passwords, and predicts the remaining suffix. The expected likelihood of the prediction increases with the length and frequency with which the prefix was observed when the model was built.

To search quickly through a large prefix of known strings and their frequencies, we use the space-efficient completion trie of Hsu and Ottaviano [10], as illustrated in Figure 3. The trie used by Telepathwords contains a 1.5GB English-language model derived from browser search queries and a set of passwords that occurred five times or more in the RockYou dataset. We removed all capitalization and spaces from the language model before building the trie.

Completion tries are already used for auto-completion and word-breaking applications, and these applications require algorithms that adapt to common misspellings and typos. For example, existing systems will walk a completion trie reversing the two characters at the suffix, applying a penalty to account for the fact that transpositions occur with much lower frequency than correctly sequenced characters. If the transposed prefix occurs with sufficient frequency to overcome the penalty, the system may continue to track that transposition and make predictions based on it.

Since Telepathwords uses tries to look for common strings that may begin anywhere in the query (e.g., `passw` in the query `notapassword`), we maintain a window of completion-trie nodes for each possible starting position, as illustrated in Figure 4. We track the trie node for each possible suffix of the query. In addition, we maintain two special windows: one that walks the trie only when letters are typed and one that does so only when digits are typed. These special windows help to detect words broken up by non-alphabetic characters (e.g., `pa1234ssword`) or numbers broken up by non-digits (e.g., `12x34y678z9`).

In contrast to other applications of tries, users choose passwords with the deliberate goal of creating a string that is hard to predict, leaving many more anomalies to detect and work around than if divergences from known strings occurred only by accident. We thus maintained a large list of windows for each queried password prefix so as to preserve nodes that might not immediately appear

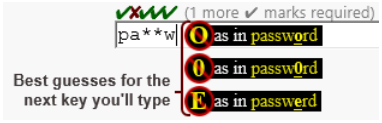


Figure 5: The **common character sequence** predictor walks the ancestry chain to see if a completion that was broken by an unpredicted character might still provide the best guess for what happens next. For `pa**w`, the third ancestor (`pa`) predicted the `w` in the fifth position, and since there are no other likely predictions, predictions using this ancestor reach the top.

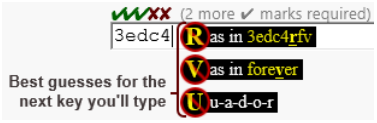


Figure 6: The password `3edc4`, composed of vertical columns on a QWERTY keyboard (`3edc`, `4rfv`, etc.), triggers the **keyboard-movement predictor** yielding `r` as the top guess for the next character. The second prediction guesses that the `4` is used in place of `for` in `forever`, and the third prediction guesses that `ecuador` is interleaved into every other character of the password.

valuable but might prove predictive as more characters arrive.

We also built a table mapping common character substitutions, such as `3` for `e`, `$` for `s`, and `0` for `o`, that are often provided in password-creation guidance (in our view, misguidedly). If we detect a character that is often substitute for another, we create a window using the character we believe was substituted for and assign that window an appropriate penalty.

To detect when users type distractor characters in place of predicted characters, then carry on with the predicted string, we walk up the ancestry path of the current prefix to look for predictions that may have been abandoned due to such behavior. For example, if the user has typed the prefix `pa**w`, the algorithm will walk up from `pa**w` to the ancestor prefix `pa`, determine that the prediction of `password` for this prefix would have correctly predicted the `w` in the fifth position, and may thus revive that prediction to predict a `o` in the next position. See Figure 5. Similarly, we use the standard error-correction technique of detecting when a user has skipped a key and typed the second character predicted in place of the next character predicted.

The analysis of each password prefix of length n begins with the analysis of its immediate prefix of length $n-1$. Thus, the cost of analysis grows at least linearly with the length of the password. We maintain a main-memory cache of recently analyzed query strings so that results can be re-used when the suffixes of a previously-queried string are queried.

Even under heavy load, the cache is small in comparison to the 1.5GB language corpus. In our deployment, the language corpus is stored in main memory. During development, we found performance to be sufficiently fast using a solid state drive (SSD) to store the corpus and only mapping pages into main memory on demand. In our deployment, we prefetched the full corpus into DRAM as our servers did not have SSDs.

Figure 7: The start of a repeating string triggers the **repetition predictor**.

2.3.2 Keyboard movements

We developed this predictor to detect passwords composed of a sequence of characters typed by moving one's finger over a sequence of adjacent keys.

We built a keyboard model that maps characters to x and y coordinates that represent the column and row of the key used to type each character on a keyboard. We represent an n-character password prefix as a sequence of n key positions, then generate a series of $n-1$ movements from the first to the last character. We then work backward from the end of the prefix to count the number of consecutive moves that are to adjacent keys and, of those, the number of consecutive moves in the same direction. We count movements that wrap from one end of the keyboard (e.g., from top to bottom) as adjacent.

We have currently mapped only QWERTY keyboards, but the implementation is generalized to support any mapping of characters to coordinates.

2.3.3 Repeated strings

This predictor looks for instances of repeated strings in password prefixes. For each possible suffix of length n, it looks for repeated sequences of the suffix. The longer the repeated sequence, the stronger the prediction. If the repetitions are adjacent to each other (`xyabcabcabc`), then the predictor guesses the next character in the repeated sequence (or the first if the end has been reached). If the suffix and its copy are not adjacent, then the early copy and the intervening string are assumed to be in the process of repeating. For example, in `abcdefabc` the suffix `abc` is repeated twice and the predictor guesses that `def` will come next.

2.3.4 Interleaved strings

This predictor looks for passwords composed of two predictable strings interleaved with each other, such as `p*a*s*s*w*o*r*d` or `ppaasswwoorrdd`. It splits passwords to separate the odd- and even-indexed characters and runs the other predictors (with interleaving-detection turned off) on the substrings. If, for example, the next character is at an even-index, it uses the even-index substring to make the prediction, and also examines the predictability of the odd-index substring in evaluating the likelihood that the query actually represents two interleaved strings.

2.4 Telemetry

Our public deployment of Telepathwords maintains a limited log of user behaviors, including page loading, resizing, key-up events, and prediction rendering events. Unless users explicitly opt-in to 'donate' their keystrokes to science, we record the timing of keyup events, the number of keys added or deleted, and the position of the change, but not the actual keys typed. We also record whether characters currently in the password field were among those predicted, recording data similar to that which is displayed in the feedback bar.

While we store logs online, the server is unable to read their contents. At the start of a user session the client-side JavaScript requests a one-time session-encryption key from the server. The server generates the key, encrypts it with a public key, and then writes the encrypted session key to the first entry of the log for the session. It then sends the key to the client and maintains no further record of it. The private key is not stored on any publicly facing server. The client XORs the log data stream with a bit stream generated by using AES in counter mode with the Stanford Javascript Crypto Library (SJCL) [27]. We opted for this approach, inspired by Kelsey and Schneier [23], because of its simplicity and as concerns over confidentiality far outweighed that of integrity. As logs are never read online, and no action is taken with them but to store them, we do not know of a scenario in which an adversary could learn the contents of the logs by modifying them.

2.5 System Limitations

The current deployment of Telepathwords has some limitations that are inherent to research prototypes. The language corpus is US-centric and somewhat dated, and so unlikely to pick up on words or phrases uncommon in the United States or that have entered the common lexicon since 2012. An ideal set of corpora would be international and receive constant updates from the latest search queries, news, and other topical sources.

Telepathwords cannot currently detect reversed character sequences (`gfedcba` in place of `abcdefg`) unless that reversal is itself already common enough to be in the language corpus (as it is for `drowssap`, for example). One way to implement reversal detection would be to reverse the more common strings in our language corpus, assess a penalty for the reversal, and insert them into our completion trie.

The privacy promises made by the current deployment of Telepathwords prohibit analysis of passwords for any purpose other than the issuing of predictions, and so the language corpus, scoring rules, and known set of common password-creation behaviors do not grow over time. Thus, if users flock to common behaviors in response to Telepathwords (as they do in response to password-composition rules) we may not be able to detect these behaviors in the current deployment.

3 Deployment

Our first deployment of the Telepathwords technology is a password-testing website, similar in purpose to existing websites that offer to test the 'strength' of passwords [9, 16, 20], which is hosted at https://telepathwords.research.microsoft.com. We took great pains to avoid positioning the service as measuring any form of 'strength' or 'security', as no system can be certain that any user-chosen password is truly strong or secure. There is no guarantee that a password that *appears* strong would not be predictable by an attacker with better knowledge of how certain users construct passwords.

As with any publicly facing Internet service, we deployed Telepathwords with some trepidation not knowing what usage levels to expect and not knowing what factors we may have failed to anticipate when performing load-testing experiments. In our pre-deployment throughput tests, Telepathwords processed 454,486 passwords in a database of breached Yahoo! Voices passwords in under 7 hours using 3 cores of a 3Ghz Xeon E5 1607 (roughly five passwords per core-second.)

We opened up our system to the public on December 5, 2013 and saw our highest usage rates shortly afterward, as the technical press published articles about the release.

3.1 Data collected

We downloaded our encrypted logs to a researcher's workstation for decryption and analysis. We graph the arrival rate of users to our service in Figure 8, which illustrates the burst of traffic during initial release dissipating over time. We are also able to observe the delay experienced by users between the time they typed a key and received a prediction for what the following key

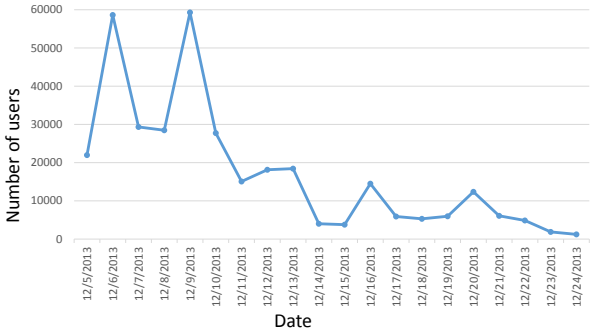


Figure 8: Sessions served per day by the Telepathwords service shortly after release. (A Session is counted when the Telepathwords page loads and the server receives a request for a session ID and encryption key used for logging.)

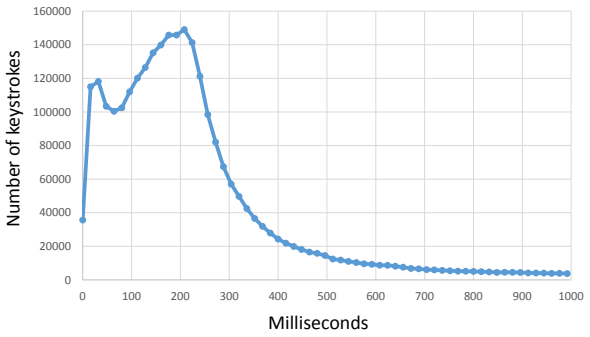


Figure 9: The distribution of the delay between the "keyup" and "render" events for all keystrokes during the recording period. The median occurs at 208ms.

would be, graphed in Figure 9. The median delay was 200ms. A peak in the graph around 20ms is likely due to fast rendering of predictions cached within the browser.

We are also able to use the logs to track how much activity users perform during each user session. In Figure 10, we examine the distribution of number of keys pressed per user session, seeing that some users appeared to use the site to test multiple passwords.

4 Experimental Methodology

In addition to the deployment, we conducted a comparative evaluation of Telepathwords and a number of existing password-composition policies via a two-part online study using Amazon's Mechanical Turk crowdsourcing service. To facilitate comparisons with prior work, much of our methdology mirrors that of a recent line of research from Carnegie Mellon University, including that of Kelley [12], Komanduri [13], Mazurek [15], Shay [24, 25], Ur [28], and others.

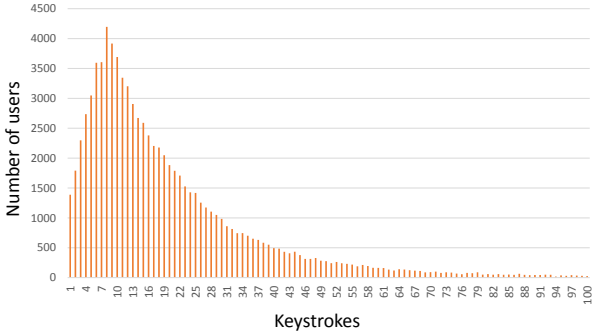


Figure 10: The number of keystrokes received per user session provides insight into user engagement with the site. The median is 15 and the mean is 21 keys pressed per session.

Our experiment was approved by Carnegie Mellon University's institutional review board prior to the start of our study.

4.1 Recruiting and Data Collection

We recruited participants from Amazon's Mechanical Turk by listing a Human Intelligence Task (HIT) in which we offered 55 cents to "Take a 5-minute survey with 70-cent bonus opportunity!" We required participants be 18 years of age and located in the United States.

We asked participants to imagine that their email account had been compromised and that they needed to create a new password to replace it. We used a round-robin algorithm to assign participants one of six password-composition policies. As users typed their proposed password, we provided real-time feedback indicating the conditions that needed to be met for participants to satisfy their assigned policy. Whereas prior CMU studies checked compliance with password policies after participants had submitted them, in this study we enabled the submit button only after a participant had satisfied the policy (and correctly retyped the password).

After participants submitted the password, we presented a survey with up to 24 questions to ask about their experience creating the password, more general questions about their password habits, and their demographics. Following the survey we asked participants to recall their passwords, giving them five attempts to do so. We displayed their password to them if they could not recall it within those five attempts. This concluded *part one* of our study.

Two days later, we invited participants to return for *part two* of our study, sending them an email via an interface provided by Mechancial Turk. We offered 70 cents to return for this HIT, in which we asked participants to recall their passwords. Again, we allowed par-

ticipants five attempts to provide the correct password. We displayed participants' passwords if they were unable to succeed within five attempts, though we did not tell them this a priori. We wanted participants to complete the study whether or not they recalled their password, so we provided them with a last-resort mechanism for recovering their passwords: a link, which would send an email, which contained a link, which led to a webpage, which displayed the correct password. We took this intentionally circuitous approach, rather than simply showing participants their passwords on request, to discourage them from using the recovery mechanism without first trying to recall their passwords. Outside the extra effort for password recovery, we did not further penalize participants for failing to recall their passwords; if we had, and future participants learned about it, they might have been more likely to store their passwords.

Finally, we asked participants to take an 18-question survey asking about their password-recall process and whether they had stored their passwords.

Except as noted, we focus our analysis on those participants who finished the first part of our study. Our analysis of dropout rates examines all participants who begin the study, and our analysis of part two examines only those participants who finished part two. We exclude participants from part two if they did not complete it within three days of the invitation.

4.2 Treatments

The only features of our study that varied between participants were the assigned password-composition policy, whether the password field hid the characters typed into it, and a few survey questions about policies specific to certain treatments. Of the six password policies we assigned to participants, two use a Telepathwords-based policy and four use policies based on composition-rules and (in one case) dictionary checks.

- **telepath, telepath-v** These two conditions employed a Telepathwords-based policy that required users to provide a password with at least six characters that were not predicted by the system. The system does not predict the first character, and so the first character of each password always counted toward the requirement. The two conditions differed only in that passwords would be shown by default as they were being typed in telepath-v and were hidden by default in telepath.
- **basic8** This condition required passwords of at least eight characters in length.
- **3class8** This condition also required passwords of at least eight characters in length, adding the requirement that the password include three of four character classes: uppercase letters, lowercase letters, digits, and symbols. This policy mirrors the default password policy for Microsoft Windows Active Directory.
- **3class12** This condition required passwords of at least 12 characters in length from three of four character classes.
- **3class8-d** This condition required passwords to include at least eight characters, from three of four character classes, and required that the string of all letters within the password not match any of the roughly 3M words in the free Openwall cracking dictionary [5].

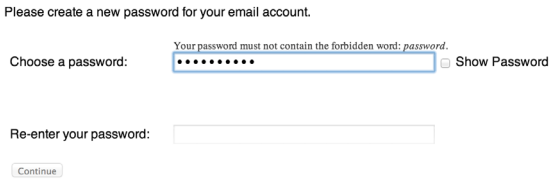


Figure 11: The 3class8-d treatment on the experimental website.

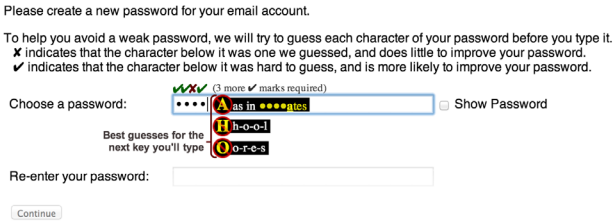


Figure 12: The telepath treatment on the experimental website.

We displayed the requirements that had not yet been met directly above the password-entry field, as shown in Figure 11. If the password had not yet met the length requirement, we displayed that requirement. If a password met the length requirement, we displayed remaining composition requirements, if any. If the password met the length and composition requirements but failed a dictionary check (for 3class8-d), we displayed the match and indicated that the password must not contain the matched word.

We displayed a checkbox that allowed participants to show or hide the characters being typed. With the exception of telepath-v, the password was hidden by default.

	basic8	3class8	3class12	3class8-d	telepath	telepath-v
Participation						
arrived at part one	476	475	472	469	476	476
finished part one	431/476 (91%)	440/475 (93%)	425/472 (90%)	402/469 (86%)	420/476 (88%)	442/476 (93%)
returned & finished part two in <3 days	270/431 (63%)	296/440 (67%)	277/425 (65%)	260/402 (65%)	267/420 (64%)	257/442 (58%)
Password Selection & Handling among part-two participants						
did not store	172/270 (64%)	197/296 (67%)	168/277 (61%)	155/260 (60%)	168/267 (63%)	141/257 (55%)
did not re-use	221/270 (82%)	228/296 (77%)	226/277 (82%)	214/260 (82%)	229/267 (86%)	203/257 (79%)
did not store or re-use	135/270 (50%)	149/296 (50%)	140/277 (51%)	118/260 (45%)	138/267 (52%)	112/257 (44%)
Password Recall in 5 tries without reminder						
during part one	423/431 (98%)	434/440 (99%)	414/425 (97%)	391/402 (97%)	407/420 (97%)	429/442 (97%)
all part-two participants	176/270 (65%)	213/296 (72%)	186/277 (67%)	193/260 (74%)	183/267 (69%)	178/257 (69%)
part two did not store	105/172 (61%)	131/197 (66%)	104/168 (62%)	103/155 (66%)	103/168 (61%)	86/141 (61%)
part two did not re-use	144/221 (65%)	163/228 (71%)	151/226 (67%)	155/214 (72%)	159/229 (69%)	143/203 (70%)
part two did not store or re-use	83/135 (61%)	97/149 (65%)	86/140 (61%)	73/118 (62%)	84/138 (61%)	69/112 (62%)

Table 1: We tally the set of participants who began part one of our study, finished it, and who returned for part two. We measure recall rates for part one (shortly after password selection) and part two. We break down part-two recall rates to factor out participants who reported re-using passwords they already knew or storing their passwords.

5 Experimental Results and Analysis

The application of multiple statistical tests increases the chance of producing a Type I error, finding a significant difference where none exists. To compensate for this, we use a standard two-step process. First, we only perform pairwise tests if an omnibus test is significant. We use the Kruskal-Wallis omnibus test (KW) for quantitative data and the χ^2 test for categorical data. Second, we correct all pairwise tests using the Holm-Bonferroni method (HC). We use the Mann-Whitney U for quantitative pairwise comparisons and Fisher's Exact Test and the χ^2 test for categorical pairwise comparisons.

We performed our experiment in February 2014. We recruited 2,844 workers to accept our HIT for part one of our study. Of these, 2,560 finished, received payment, and received invitations to return two days later. A total of 1,627 participants (64%) returned for the second HIT within three days of when we sent their invitation. Participants' demographics reflected a typical population of workers on Mechanical Turk, with a median reported age of 27, nearly 60% reporting as male, and 44% reporting having at least a bachelor's degree.

We show the progress of participants through our study in Table 1. We removed from our analysis five participants who created more than one password by using the back button or reloading the password-creation page.

The condition with the highest dropout rate was 3class8-d, while the lowest dropout rate was telepath-v. Table 2 shows the dropout rates for each condition. It also gives the test's p-value for the null hypothesis (that the difference between dropout rates was unaffected by condition) for each pair of conditions. For example, at $p = 0.01$ (resp. $p = 0.007$) the difference in dropout rates between 3class8-d and 3class8 (resp. telepath-v) is significant. For all of the other condition pairs the hypothesis that condition had no effect on dropout rate is not ruled out. Note that the table has a triangular structure since we list the result of each pairwise test only once, and this same format is used for our other categorical tests (i.e. Tables 3, 4, 5, 6 and 7).

			Fisher's Exact Test p (Holm-Bonferroni corrected)				
Treatment	Part one dropout		telepath	3class12	basic8	3class8	telepath-v
3class8-d	67/469	14%	1.000	.458	.318	.010	.007
telepath	56/476	12%		1.000	1.000	.318	.255
3class12	47/472	10%			1.000	1.000	1.000
basic8	45/476	9%				1.000	1.000
3class8	35/475	7%					1.000
telepath-v	34/476	7%					

Omnibus χ^2_5=19.373, p=0.002

Table 2: The fraction of participants who dropped out during part one, with corrected pairwise comparisons of all treatment groups.

The median time spent to create a password was 32 seconds for participants in basic8, 43 for 3class8, 53 for both 3class12 and 3class8-d, 85 for telepath-v, and 96 for telepath. We anticipated participants using Telepathwords might spend more time, as these treatments included three lines of instructions not present in other treatments (see Figure 12) and their novelty may have led to more exploration.

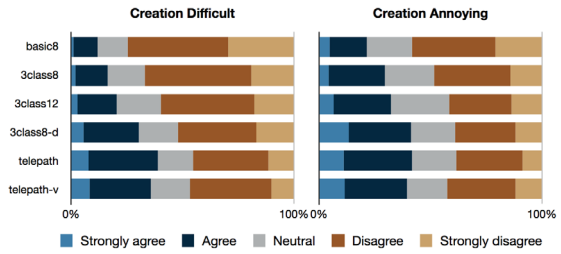


Figure 13: "Creating my password was difficult" and "Creating my password was annoying."

5.1 Recall

We provide recall rates for part one of the study for reference only, as just minutes had passed since participants had chosen their passwords. In the second section of Table 1, we see that 61.5% of participants indicated that they had not stored their password and that we had not detected them pasting or auto-filling a password into the recall field. Differences between treatment groups were not statistically significant (χ^2_5=9.231, p=0.1).

We looked at the number of part two recall attempts by the subset of participants who did not store their passwords, did not use the reminder feature, and did not re-use a previous password. Of these 502 participants, 79.3% entered the password on the first attempt, and 14.5% entered it on the second attempt. While the omnibus test shows a significant difference between conditions for taking more than one attempt (χ^2_5=13.943, p =0.016), the pairwise tests showed no significant differences. Among the 398 of these participants who entered their password correctly on the first attempt, the median password-entry time was 14.8 seconds; this did not vary significantly by condition (KW χ^2_5=4.705, p =0.453). Looking at the 1159 participants who did not use the reminder, 80.9% entered the password correctly on the first try. This differed by condition (χ^2_5=12.604, p =0.027), but no pairwise test was significant.

5.2 Participant Sentiment

In addition to recording participant behavior, we asked participants about their experience. We asked all participants whether they felt that creating their password was difficult or annoying, with results in Figure 13. We show the pairwise comparisons across conditions for difficulty in Table 3 and annoyance in Table 4.

The three policies that tested participants' passwords against lists of common passwords (the Telepathwords conditions and 3class8-d) had a greater proportion of participants who were annoyed than those using the purely composition-based policies. The differences with the simplest policies were significant, as shown in Table 4.

			Fisher's Exact Test p (Holm-Bonferroni corrected)				
Treatment	Creation difficult		telepath-v	3class8-d	3class12	3class8	basic8
telepath	163/420	39%	.374	.078	<.001	<.001	<.001
telepath-v	158/442	36%		.374	<.001	<.001	<.001
3class8-d	123/402	31%			.006	<.001	<.001
3class12	87/425	20%				.374	.005
3class8	73/440	17%					.209
basic8	51/431	12%					

Omnibus χ^2_5=135.199, p<.001

Table 3: The fraction of participants in each treatment who agreed that it was difficult to create a password during the experiment.

			Fisher's Exact Test p (Holm-Bonferroni corrected)				
Treatment	Creation annoying		3class8-d	telepath-v	3class12	3class8	basic8
telepath	175/420	42%	1.000	1.000	.034	.002	<.001
3class8-d	165/402	41%		1.000	.052	.004	<.001
telepath-v	175/442	40%			.117	.013	<.001
3class12	136/425	32%				1.000	.005
3class8	129/440	29%					.052
basic8	92/431	21%					

Omnibus χ^2_5=61.805, p<.001

Table 4: The fraction of participants in each treatment who agreed that it was annoying to create the password during the experiment.

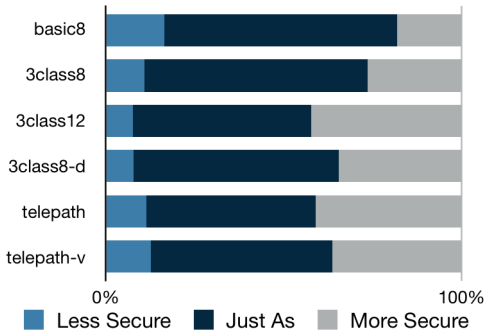


Figure 14: "When compared to the password I use for my primary email account, the password I created for this study was:".

We also asked participants whether they believed their study-created password to be more, less, or just as secure as their primary email password. The results are in Figure 14. Belief that the study passwords were more secure

Treatment	More Secure		Fisher's Exact Test p (Holm-Bonferroni corrected) telepath	telepath-v	3class8-d	3class8	basic8
3class12	180/425	42%	1.000	.329	.134	<.001	<.001
telepath	172/420	41%		.552	.309	<.001	<.001
telepath-v	161/442	36%			1.000	.013	<.001
3class8-d	139/402	35%				.075	<.001
3class8	116/440	26%					.027
basic8	78/431	18%					

Omnibus χ^2_5=83.62, p<.001

Table 5: The fraction of participants who selected "More secure" in response to "When compared to the password I use for my primary email account, the password I created for this study was:".

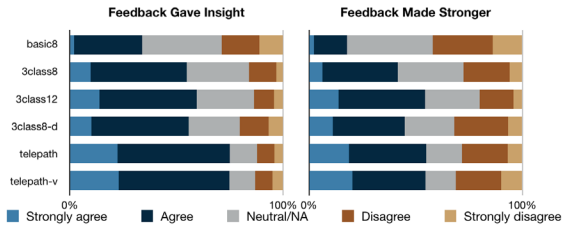


Figure 15: "The visual feedback I received gave me insight into the quality of my password" and "The visual feedback that was displayed helped me to create a stronger password that I would have otherwise."

ranged from 18.1% for basic8 to 42.4% for 3class12, and significant differences are in Table 5.

We displayed visual feedback in *all* conditions to help participants comply with their assigned policy. We asked participants if they believed the feedback gave them insight into their passwords and if it helped them to create better passwords. We show their responses in Figure 15 paired with significance tests in Tables 6 and 7.

The Telepathwords treatments, along with 3class12, had the greatest proportion of participants who believed the feedback helped them create a stronger password. A significantly larger portion of participants in the two Telepathwords conditions agreed that the feedback provided more insight than the other treatments—including the dictionary-based feedback in 3class8-d. This is tempered of course by the higher number who found password creation difficult or annoying with the tool. We see this as a hopeful sign that Telepathwords can help improve the credibility of technology designed to prevent users from choosing weak passwords.

Treatment	Gave Insight		Fisher's Exact Test p (Holm-Bonferroni corrected) telepath-v	3class12	3class8-d	3class8	basic8
telepath	315/420	75%	1.000	<.001	<.001	<.001	<.001
telepath-v	331/442	75%		<.001	<.001	<.001	<.001
3class12	253/425	60%			.873	.678	<.001
3class8-d	224/402	56%				1.000	<.001
3class8	241/440	55%					<.001
basic8	146/431	34%					

Omnibus χ^2_5=208.104, p<.001

Table 6: Agreement with "The visual feedback I received gave me insight into the quality of my password."

Treatment	Feedback Helped		Fisher's Exact Test p (Holm-Bonferroni corrected) telepath-v	3class12	3class8-d	3class8	basic8
telepath	231/420	55%	1.000	1.000	.029	.001	<.001
telepath-v	241/442	55%		1.000	.029	.001	<.001
3class12	231/425	54%			.033	.001	<.001
3class8-d	180/402	45%				1.000	<.001
3class8	183/440	42%					<.001
basic8	77/431	18%					

Omnibus χ^2_5=178.62, p<.001

Table 7: Agreement with "The visual feedback that was displayed helped me to create a stronger password that I would have otherwise."

5.3 Security Results

In Table 8 we present statistics summarizing the composition of passwords created under each policy, and security scores calculated by three metrics. We focus our analysis on the passwords identified to be weakest as an attacker is most likely to try these first. Dictionary attacks to obtain beachheads into organizations succeed when the first account is breached. Thus, improving the security of the weakest password in an organization by a small amount is far more likely to prevent an attacker from obtaining a beachhead than a large improvement to the average password would. This is particularly true for an online attack where a limited number of guesses per account can be tried.

We did not encounter any repeat passwords in our sample, so we cannot use frequency as a metric. Rather, the first metric we apply is an entropy calculation generated by the open-source `zxcvbn` password meter [30]. Its advantages are that it is publicly available, open-source, and already relied on by large-scale systems, including DropBox. Its primary disadvantage is that it was de-

signed to meet the constraints required for deployment as a client-side password meter; it needed to be small enough to download quickly and efficient enough to run in JavaScript. As such, it cannot perform the same level of computational analysis or apply the same body of knowledge as a tool designed for guessing.

The second metric we apply is a guess-number calculator developed by Saranga Komanduri, which first appeared in Kelley *et al.* [12, 25]. We call this metric Weir+ because it builds on the guessing approach of Weir et al. [29]. Its advantages are that it is designed with the explicit goal of measuring the number of guesses required to crack a password, can be trained to target specific password policies, and represents the state of the art in measuring strength against a guessing attack. The disadvantages of Weir+ include that it is available only by contacting the author, written in multiple programming languages, and has not been made easy to configure. Further, its results may vary based on the size and quality of the training data. In order to create a large training set of passwords that comply with the Telepathwords policies, we used the 133,109 passwords in the Yahoo! Voices breach data set that received a score of 6 hard-to-guess characters or more from Telepathwords, which represents 29% of the 453,488 passwords revealed by that breach.

Our final metric is the score provided by the current version of Telepathwords itself—the number of hard-to-guess characters. We find this informative for comparing treatments *other* than those that employ Telepathwords. The scores for participants in telepath and telepath-v are provided exclusively for completeness, as participants who were able to generate a password that met the Telepathwords policy will score well by default (though we note that two participants received a 5 due to a change to predictions from the version deployed during the experiment and the version used to calculate scores).

Regardless of metric, the telepath and telepath-v passwords do substantially better than all other conditions, with the possible exception of 3class8-d. We present the scores for each metric in Table 8.

For the `zxcvbn` entropy measure, we show in Figure 16 that telepath and telepath-v passwords outperform those from all other conditions for the weakest password in each condition and the weakest 2.5%, 5%, and 10% of passwords. Thus, Telepathwords did the best job of preventing weak passwords. Only when we consider the median entropy do 3class12 and 3class8-d become competitive. The improvement with respect to 3class8 and basic8 is enormous.

Figure 17 illustrates the Weir+ measurements. Again the two Telepathwords conditions show enormous improvement over basic8 and 3class8. They show considerable improvement over 3class12 on minimum entropy, and on entropy of the weakest 2.5%, 5%, and 10%. The 3class8-d condition is roughly comparable to the two Telepathwords conditions, except when we consider minimum entropy, where it does considerably worse.

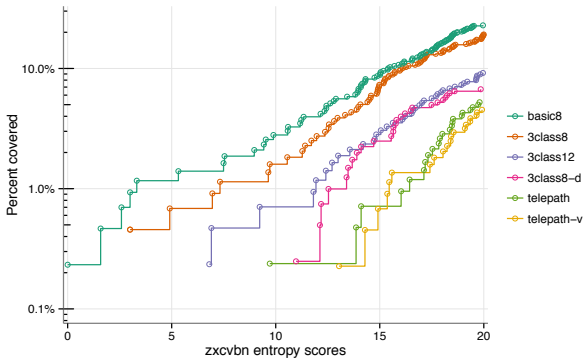


Figure 16: We sort the passwords in each condition by `zxcvbn`-entropy scores, from lowest to highest, and present the fraction of passwords with scores at or below a given value. Only passwords with entropy scores of 20 or less are shown in order to highlight the weakest passwords in each condition.

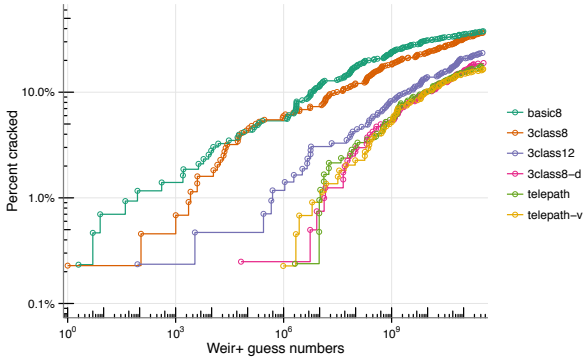


Figure 17: We sort the passwords in each condition by their Weir+ guess number, from lowest to highest, and present the fraction of passwords that with guess counts at or below a given number of guesses.

To substantiate further the impact of using Telepathwords and dictionary-based approaches, we present in Table 9 the weakest 2.5% of passwords according to each metric. For example, the weakest 2.5% under 3class8 contain such obvious and easily-guessed choices as `Password1` and `P@5sword`, which compare unfavorably with those in either of the Telepathwords conditions of 3class8-d.

		Mean characters per class				zxcvbn Entropy					Telepathwords score					Weir+ score			
	N	Upper	Lower	Digit	Symbol	Min	2.5%	5%	10%	Med	Min	2.5%	5%	10%	Med	Min	2.5%	5%	10%
3class12	425	1.7	8.0	3.2	0.9	6.8	14.8	17.0	20.4	33.4	2	3	3	4	7	6.4	22.4	27.8	31.5
3class8-d	402	1.5	6.6	2.6	0.8	11.0	15.6	17.8	22.1	32.7	1	3	3	3	6	16.0	26.6	29.4	33.6
3class8	438	1.6	6.6	2.5	0.6	3.0	11.7	14.1	16.1	29.1	1	2	2	3	6	0.0	14.4	17.4	24.9
basic8	429	1.0	7.9	2.4	0.4	0.0	9.5	12.6	15.4	27.9	1	2	2	3	6	1.0	13.0	17.7	22.4
telepath	420	1.0	7.0	2.5	0.6	9.7	17.9	19.7	22.4	32.0	5	6	6	6	7	21.0	26.1	29.3	33.0
telepath-v	441	1.1	7.0	2.7	0.5	13.0	18.4	20.4	22.4	32.8	6	6	6	6	7	19.9	27.4	29.9	33.2

Table 8: Security metrics of passwords created by participants. We show minimum and median zxcvbn and Telepathwords scores, along with percentiles selected to indicate the vulnerability of each condition to early guessing. We report Weir+ scores as the $\log_2$ of their guess numbers for comparison with entropy scores. We do not show median Weir+ scores as only basic8 reached 50% cracked in our analysis.

zxcvbn					
basic8	3class8	3class12	3class8-d	telepath	telepath-v
password	Password1	Thispassword1	1qaz2wsx!	thisisapassword	guessmypassw0rd
12345678	P@5sword	Password@123	123456789jI	2014welcome	Mary3476
P@55w0rd	EL1Z@B3TH	Qwerty12345@	Zaq12wsx	jim1965	altoids123
PASSWORD1234	Password8	Passwordneeds1	@bs0lute	$hrod3	almay123
passwordme	Mypassword1	!PaSsWoRd123	A11iance	1024scott	the1step!
sunshine	Samantha1	StephenASmith1	Beer4y0u	mothertrucker	snoopy1969!
Youknow123	Whatever1	1NewP@ssword	Hawk3y3s	burkeds	kylemonkey1986
brittany	Whoi1234	Chief$123456	G0dZ1Ll4	pi$$a123	Scr3wdr1v3r123
drowssap	My2password	MonKeY12345!	@SunSh1n3	12noraa	sion12
Washington1	Shelby1234	Asdfghjkl123	Cut13p13	c@reful951	lmi2014
					1987camaros
Weir+					
password	Password1	Asdfghjkl123	Pokemon91	1024scott	iamabeliever
12345678	Password8	Password@123	Redtruck1	jim1965	feefifofum
sunshine	Rainbow3	bulldog*1234	Nackson1	cesar5000	motuwethfr
brittany	Robert07	Jp1234567890	ZaqXsw12	mi1213	snorelax
qwertyuiop	Cougars1	Johnny#12345	H1r12345	mothertrucker	broseph
drowssap	Andrew24	Strawberry246	Monkeydude1	thisisapassword	cats59
trinity1	Marcus12	Guadalajara1	Plascencia1	imalittleteapot	peacaboo1
sugarbaby	Liverpool15	123Cheetos!!	Caedus12	awdxsz	almay123
deeznuts	Bahamut1	Abc123456789!	Godalmighty1	chieri	altoids123
monkey69	Abby1234	Qwerty12345@	Yaniku13	coffeecup123	jacran1
					sion12
Telepathwords					
frenchfry	Password1	MountainDew1	BearBear1		
qwertyuiop	EL1Z@B3TH	P00lsidebars	B4sk3r*v1ll3		
password	P@5sword	Elephants.19	Redtruck1		
p09op09o	Robert07	Password@123	A11iance		
P@55w0rd	Samantha1	cRAYON123456	Ilove!myself		
PASSWORD1234	Whatever1	MonKeY12345!	Zaq12wsx		
R0ckstar!	Qwaszx12	Abc123456789!	Cut13p13		
	Monkeys21	!PaSsWoRd123	Monkeydude1		
	Scoobydoo2	Asdfghjkl123	ZaqXsw12		
		Qwerty12345@	Galvestontx1		

Table 9: The weakest 2.5% of passwords as scored by each metric (weakest at top). For the Telepathwords metric, the weakest passwords in the telepathwords conditions are not shown because there are too many passwords at the minimum score threshold to present here.

5.4 Limitations

All artificial experiments have limitations and ours was no exception. We make note of two such limitations.

Our study used a role-playing scenario to encourage users to create passwords. Participants playing roles may choose weaker passwords than they would for an account they value. They might also choose stronger passwords than they would for an account they didn't value. Schechter *et al.* [22] have shown that participants in security studies behave differently when the laboratory en-

vironment frees them from risk, and Fahl *et al.* [7] have shown a specific effect for choice of passwords. While limiting the interpretations of absolute scores, so long as these effects impact conditions equally, the methodology still facilitates cross-condition comparisons—the primary focus of the experiment. In fact, if our goal is to study the ability of technology to help unmotivated users choose better passwords, having participants who are less motivated than they would be in real-world conditions may be beneficial.

We measured recall over a short period of two to five days in a context where participants entered their password a few minutes after choosing it. In contexts in which users do not re-enter their passwords immediately after creating them, or in which they do not return for more than five days, they may be more likely to forget them. In contexts where users use their passwords more frequently after creating them, they may be less likely to forget them within two to five days. Had we selected different return periods we might have been more likely to see differences in recall rates.

6 Related Work

While some security practitioners simply hope that passwords, and their associated weaknesses, can be wished away, Bonneau *et al.* [3] have argued that passwords are not going away anytime soon. Password-composition rules date back at least to 1979, when Morris and Thompson reported on the predictability of the passwords used by users on their Unix systems; they proposed that passwords longer than four characters, or purely alphabetic passwords longer than five characters, will be "very safe indeed" [19]. Bonneau analyzed nearly 70 million passwords in 2012, 33 years later, to measure the impact of a six-character minimum requirement compared with no requirement [2]. He found that it made almost no difference in security. In a study of the distribution of password policies, Florêncio and Herley found that usability imperatives appeared to play at least as large a role as security among the 75 websites examined [8].

Early studies of proactive password-quality verification mechanisms includes the work of Spafford [26], who suggests an efficient method for storing a dictionary for checking. Bishop *et al.*, in 1995, suggested checking passwords for dictionary entries, user information, and other common patterns at password creation [1]. They also provided some statistics on these patterns in passwords. Weir *et al.* also examined password-composition rules by looking at samples of passwords [29]. These works did not look at passwords created under varying rules, however.

Microsoft Windows has enforced password-composition rules at least as far back as 2000, with the default requiring at least 8 characters from three of four character classes: uppercase, lowercase, digits, and others [17, 18]. One problem with the Windows implementation is that when Windows rejects a user's proposed password, it does not provide a list of the rules being enforced or identify specifically which rules the password is violating.

Many websites offer password meters that provide feedback on the strength of passwords as users type them. Based on a survey of the top 100 websites in 2012, most password meters use simple password-composition rules such as length and number of non-lowercase characters to determine when a password is good enough to reach the next level on the meter [28]. Egelman *et al.* [6] examined whether the presence of a password meter made any appreciable difference in password strength. They found that the meter made a difference when users were changing their password for an existing important account; but the meter had little effect when users were registering a new password for a low-importance account. Ur *et al.* also studied the effect of password-strength meters on password-creation. They found that when users became frustrated and lost confidence in the meter, more weak passwords appeared [28]. Very recently, de Carné de Carnavalet and Mannan [4] examined several password meters in use at popular websites and found gross inconsistencies, with the same password registering very different strength across different meters. Collectively, these findings are in line with our concern that password policies and meters may harm credibility and lead users to put less effort into choosing a good password.

One exception to the reliance on composition rules in password meters is `zxcvbn`, an open-source meter developed and used by DropBox, which uses a small language corpus to calculate entropy estimates in real time [30]. Designed to run entirely in the users' browser, it is written in JavaScript and compresses down to 320KB. While `zxcvbn` provides a much-needed improvement in the credibility of its strength estimates when compared to approaches relying solely on composition rules, this credibility is unlikely to be observed by users. In fact, its perceived credibility may suffer if users, who have been told that adding characters increases password strength, see scores decrease when certain characters are added. For example, when typing `iatemylunch`, the strength estimate decreases from the second-best score (3) to the worst score (1) when the final character is added. Even if users find `zxcvbn`'s strength estimates credible, they are unlikely to understand the underlying entropy-estimation mechanism and thus be unsure how to improve their scores. The advice `zxcvbn` offers, such as using inside jokes and unusual use of uppercase, could potentially

lead users to cluster around common strategies, yielding a set of new common passwords for attackers to guess.

Schechter *et al.* [21] offered another alternative to password-composition rules, suggesting a system that prevents users from choosing passwords popular among a large set of users. Another approach that seeks to limit dangerously common passwords was proposed by Malone and Maher [14]. These approaches, however, are most appropriate for systems with tens of millions of users, in which uniqueness is a strong indicator that a password is hard to guess. Relatively weak passwords may be unique among hundreds or thousands of user accounts.

The human-subjects experiment we perform in this work seeks to replicate the methodology used in prior password studies. Many of our choices in recruiting, question design, and the timing of the invitation to part two of the study reflect a desire to facilitate comparison with prior work. This includes the work of Komanduri *et al.* [13] and Kelley *et al.* [12], who used similar study designs to perform comparative analyses of password-composition rules. These prior studies found that increasing length requirements in passwords generally led to more usable passwords that were also less likely to be identified as weak by their guessing algorithm [13, 12]. Most recently, Shay *et al.* studied password-composition policies requiring longer passwords, finding the best performance came from mixing a 12-character minimum with a requirement of three character sets [25]. One key difference between our work and most prior studies is that all of our treatments provided feedback to users as they typed their passwords. With the exception of Ur *et al.*'s examination of meters providing optional guidance [28], all of these prior studies from Carnegie Mellon required participants to submit passwords *before* testing for, or providing feedback on, compliance with a policy.

A valuable use case for Telepathwords-based policies, which do not place any character-set requirements on passwords, is the affordance of creating all-lowercase passwords for easy entry on a touch screen. Jakobsson and Akavipat proposed a scheme for mobile devices that uses easily-typed passwords with auto-completion for easier password entry [11].

Fahl *et al.* [7] pointed out limitations in studies that use role-playing to generate passwords, as we do in this study. They find significant differences between passwords generated in these scenarios and real passwords. Komanduri *et al.* found that users created stronger passwords when asked to role-play, compared to when asked simply to create a password for a study [13]. Mazurek *et al.* used a methodology similar to ours and compared their results to genuine user passwords in a university [15]. They found that while the experimental passwords were slightly weaker than the genuine passwords, they were similar in many other respects.

7 Conclusion

Telepathwords provides users with significantly more insight into the quality of their passwords than all other approaches, and results in passwords stronger than approaches that do not use dictionaries. For example, the metrics suggest that to crack 1% of Telepathwords passwords, an attacker needs to make more than a factor of a thousand more guesses per password than for passwords created under the default password policy employed by Microsoft Windows Active Directory. While a higher number of users found password creation difficult or annoying using the tool, the security improvements did not come at any measurable impact to memorability.

References

[1] BISHOP, M., AND V KLEIN, D. Improving system security via proactive password checking. *Computers & Security 14*, 3 (1995), 233–249.

[2] BONNEAU, J. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy* (May 2012).

[3] BONNEAU, J., HERLEY, C., VAN OORSCHOT, P. C., AND STAJANO, F. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *2012 IEEE Symposium on Security and Privacy* (May 2012).

[4] DE CARNAVALET, X. D. C., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Network and Distributed System Security Symposium (NDSS14)* (2013).

[5] DESIGNER, S. Openwall project free wordlist. `http://download.openwall.net/pub/wordlists/all.gz`.

[6] EGELMAN, S., SOTIRAKOPOULOS, A., MUSLUKHOV, I., BEZNOSOV, K., AND HERLEY, C. Does my password go up to eleven?: the impact of password meters on password selection. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2013), ACM, pp. 2379–2388.

[7] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security* (2013), ACM, p. 13.

[8] FLORÊNCIO, D., AND HERLEY, C. Where Do Security Policies Come From? *Proc. SOUPS* (2010).

[9] How secure is my password. `https://howsecureismypassword.net/`.

[10] HSU, B., AND OTTAVIANO, G. Space-efficient data structures for top-k completion. In *22nd International World Wide Web Conferences* (May 13–17 2013).

[11] JAKOBSSON, M., AND AKAVIPAT, R. Rethinking passwords to adapt to constrained keyboards.

[12] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LÓPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. *IEEE Symposium on Security and Privacy 0* (2012), 523–537.

[13] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 2595–2604.

[14] MALONE, D., AND MAHER, K. Investigating the distribution of password choices. In *Proc. WWW* (2012).

[15] MAZUREK, M. L., KOMANDURI, S., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., KELLEY, P. G., SHAY, R., AND UR, B. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 173–186.

[16] MICROSOFT CORPORATION. Check your passwordis it strong? `https://www.microsoft.com/en-gb/security/pc-security/password-checker.aspx`.

[17] MICROSOFT CORPORATION. TechNet Configuring Password Policies. Microsoft TechNet. `http://msdn.microsoft.com/en-us/library/cc236715.aspx`.

[18] MICROSOFT CORPORATION. Windows NT 4.0 Domain Controller Configuration Checklist. Microsoft TechNet. `http://technet.microsoft.com/en-us/library/cc722923.aspx`.

[19] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Communications of the ACM 22*, 11 (1979), 594—597.

[20] The password meter. `http://www.passwordmeter.com/`.

[21] SCHECHTER, S., HERLEY, C., AND MITZENMACHER, M. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *The 5th USENIX Workshop on Hot Topics in Security (HotSec)* (Aug. 10 2010).

[22] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The emperors new security indicators: An evaluation of website authentication and the effect of role playing on usability studies. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007).

[23] SCHNEIER, B., AND KELSEY, J. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security 2* (1999), 159–176.

[24] SHAY, R., KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., UR, B., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Correct horse battery staple: exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (New York, NY, USA, 2012), SOUPS '12, ACM, pp. 7:1–7:20.

[25] SHAY, R., KOMANDURI, S., DURITY, A. L., HUH, P. S., MAZUREK, M. L., SEGRETI, S. M., UR, B., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Can long passwords be secure and usable? In *CHI* (2014).

[26] SPAFFORD, E. H. OPUS: Preventing weak password choices. *Computers & Security 11*, 3 (1992), 273–278.

[27] STARK, E., HAMBURG, M., AND BONEH, D. Symmetric Cryptography in Javascript. In *Annual Computer Security Applications Conference* (2009), pp. 373–381.

[28] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M. L., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX Security Symposium* (Aug. 8–10 2012).

[29] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 162–175.

[30] WHEELER, D. zxcvbn: realistic password strength estimation. Dropbox Tech Blog. `https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/`, Apr. 2004.

Towards reliable storage of 56-bit secrets in human memory

Joseph Bonneau
Princeton University

Stuart Schechter
Microsoft Research

Abstract

Challenging the conventional wisdom that users cannot remember cryptographically-strong secrets, we test the hypothesis that users can learn randomly-assigned 56-bit codes (encoded as either 6 words or 12 characters) through *spaced repetition*. We asked remote research participants to perform a distractor task that required logging into a website 90 times, over up to two weeks, with a password of their choosing. After they entered their chosen password correctly we displayed a short code (4 letters or 2 words, 18.8 bits) that we required them to type. For subsequent logins we added an increasing delay prior to displaying the code, which participants could avoid by typing the code from memory. As participants learned, we added two more codes to comprise a 56.4-bit secret. Overall, 94% of participants eventually typed their entire secret from memory, learning it after a median of 36 logins. The learning component of our system added a median delay of just 6.9 s per login and a total of less than 12 minutes over an average of ten days. 88% were able to recall their codes exactly when asked at least three days later, with only 21% reporting having written their secret down. As one participant wrote with surprise, "the words are branded into my brain."

1 Introduction

Humans are incapable of securely storing high-quality cryptographic keys . . . they are also large, expensive to maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed. But they are sufficiently pervasive that we must design our protocols around their limitations.

—Kaufman, Perlman and Speciner, 2002 [54]

The dismissal of human memory by the security community reached the point of parody long ago. While assigning random passwords to users was considered standard as recently in the mid-1980s [26], the practice died out in the 90s [4] and NIST guidelines now presume all passwords are user-chosen [32]. Most banks have even given up on expecting customers to memorize random four-digits PINs [22].

We hypothesized that perceived limits on humans' ability to remember secrets are an artifact of today's systems, which provide users with a single brief opportunity during enrolment to permanently imprint a secret password into long-term memory. By contrast, modern theories of the brain posit that it is important to *forget* random information seen once, with no connection to past experience, so as to avoid being overwhelmed by the constant flow of new sensory information [10].

We hypothesized that, if we could relax time constraints under which users are expected to learn, most could memorize a randomly-assigned secret of 56 bits. To allow for this memorization period, we propose using an alternate form of authentication while learning, which may be weaker or less convenient than we would like in the long-term. For example, while learning a strong secret used to protect an enterprise account, users might be allowed to login using a user-chosen password, but only from their assigned computer on the corporate network and only for a probationary period. Or, if learning a master key for their password manager, which maintains a database of all personal credentials, users might only be allowed to upload this database to the network after learning a strong secret used to encrypt it.

By relaxing this time constraint we are able to exploit *spaced repetition*, in which information is learned through exposure separated by significant delay intervals. Spaced repetition was identified in the 19$^{\text{th}}$ century [43] and has been robustly shown to be among the most effective means of memorizing unstructured information [35, 11]. Perhaps the highest praise is its popularity amongst medical students, language learners, and others who are highly motivated to learn a large amount of vocabulary as efficiently as possible [34, 91].

To test our hypothesis, we piggybacked spaced repeti-

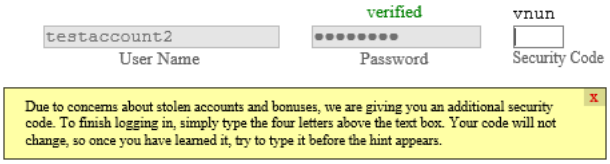


Figure 1: The login form for a user logging in for the first time, learning a code made of *letters*.

tion of a new random secret onto an existing login process utilizing a user-chosen password. Our system can be seen in action in Figure 1. After receiving a user's self-chosen password, we add a new field into which they must type a random security code, which we display directly above this field. With each login we add a $\frac{1}{3}$ second delay (up to a maximum of 10 seconds) before displaying the hint for them to copy, encouraging them to type the code from memory if possible to save time.

We recruited remote research participants to perform a study that required logging into a website 90 times over up to 15 days, which they did at an average rate of nine logins per day. We assigned each participant a random 56-bit 'security code' encoded into three chunks of either four lowercase letters or two words. After participants began to enter the first chunk before it was displayed, we added a second and likewise for the third and final chunk. We did not tell participants that learning the random secret was a goal of the research study; they simply learned it to save time. Participants experienced a median additional delay from using our system of just 6.9 s on each login, or about 11 m 53 s total over the entire study.

Three days after participants completed the initial study and had stopped using their security codes, we asked them to recall their code from memory in a follow-up survey which 88% completed. They returned after a median of 3 days 18 hours (mean 4 days 23 hours). We found that 46 of 56 (82%) assigned *letters* and 52 of 56 (93%) assigned *words* recalled their codes correctly. Only 21% reported writing down or otherwise storing the security codes outside their memory and the recall rate was actually higher amongst those who didn't.

While 56-bit secrets are usually overkill for web authentication, the most common use of passwords today, there are several compelling applications for "high value" passwords such as master passwords for password managers, passwords used to protect private keys, device-unlock passwords, and enterprise login passwords where cryptographically-strong passwords can eliminate an entire class of attack. In debunking the myth that users are inherently incapable of remembering a strong secret, we advocate that using spaced repetition to train users to remember strong secrets should be available in every security engineer's toolbox.

2 Security goals

Evaluating the difficulty of guessing user-chosen passwords is messy [56] and security engineers are left with few hard guarantees beyond empirical estimates of min-entropy, which can be as low as 10 bits or fewer [18]. By contrast, with random passwords we can easily provide strong bounds of the difficulty of *guessing*, if not other attack vectors against passwords [20].

2.1 The cost of brute-force

Random passwords are primarily a defense against an *offline attack* (eq. *brute-force attack*), in which the attacker is capable of trying as many guesses as they can afford to check computationally. We can estimate the cost of brute-force by observing the Bitcoin network [3], which utilizes proof-of-work with SHA-256 to maintain integrity of its transaction ledger and hence provides direct monetary rewards for efficient brute force. While SHA-256 is just one example of a secure hash function, it provides a reasonable benchmark.

In 2013, Bitcoin miners collectively performed $\approx 2^{75}$ SHA-256 hashes in exchange for bitcoin rewards worth $\approx$ US$257M. This provides only a rough estimate as Bitcoin's price has fluctuated and Bitcoin miners may have profited from carrying significant exchange-rate risk or utilizing stolen electricity. Still, this is the only publicly-known operation performing in excess of 2^{64} cryptographic operations and hence provides the best estimate available. Even assuming a centralized effort could be an order of magnitude more efficient, this still leaves us with an estimate of US$1M to perform a 2^{70} SHA-256 evaluations and around US$1B for 2^{80} evaluations.

In most scenarios, we can gain equivalent security with a smaller secret by *key stretching*, deliberately making the verification function computationally expensive for both the attacker and legitimate users [66, 57]. Classically, this takes the form of an iterated hash function, though there are more advanced techniques such as memory-bound hashes like `scrypt` [69] or halting password puzzles which run forever on incorrect guesses and require costly backtracking [25].

With simple iterated password hashing, a modern CPU can compute a hash function like SHA-256 at around 10 MHz [1] (10 million SHA-256 computations per second), meaning that if we slow down legitimate users by ≈ 2 ms we can add 14 bits to the effective strength of a password, and we can add 24 bits at a cost of ≈ 2 s. While brute-forcing speed will increase as hardware improves [38], the same advances enable defenders to continuously increase [72] the amount of stretching in use at constant real-world cost [19], meaning these basic numbers should persist indefinitely.

2.2 Practical attack scenarios

Given the above constraints, we consider a 56-bit random password a reasonable target for most practical scenarios, pushing the attacker cost around US$1M with 14 bits (around 2 ms) of stretching, or US$1B with 24 bits (around 2 s) of stretching. Defending against offline attacks remains useful in several scenarios.

Password managers are a compelling aid to the difficulty of remembering many passwords online, but they reduce security for all of a user's credentials to the strength of a master password used to encrypt them at rest. In at least one instance, a password management service suffered a breach of the systems used to store users' data [63]. Given that password managers only need to decrypt the credentials at startup, several seconds of stretching may be acceptable.

Similarly, when creating a public/private key pair for personal communication, users today typically use a password to encrypt the private key file to guard against theft. Given a sufficiently strong random password, users could use their password and a unique public salt (e.g., an email address) to seed a random number generator and create the keys. The private key could then simply be rederived when needed from the password, preventing the need for storing the private key at all. This application also likely tolerates extensive stretching.

Passwords used to unlock personal devices (e.g. smartphones) are becoming increasingly critical as these devices are often a second factor (or sole factor) in authentication to many other services. Today, most devices use relatively weak secrets and rely on tamper-proof hardware to limit the number of guesses if a device is stolen. Strong passwords could be used to remove trust in device hardware. This is a more challenging application, however. The budget for key-stretching may be 14 bits or fewer, due to the frequency with which users authenticate and the limited CPU and battery resources available. Additionally, entering strong passwords quickly on a small touchscreen may be prohibitive.

Finally, when authenticating users remotely, such as logging into an enterprise network, security requirements may motivate transitioning from user-chosen secrets to strong random ones. Defending against *online guessing*, in which the attacker must verify password guesses using the genuine login server as an oracle, can be done with far smaller random passwords. Even without explicit rate-limiting, attacking a 40-bit secret online would generate significantly more traffic than any practical system routinely handles. 40-bit random passwords would ensure defense-in-depth against failures in rate-limiting.

Alternately, attackers may perform an offline attack if a remote authentication server is breached. In general, we would favor back-end defenses against password database compromises which don't place an additional burden on users—such as hashing passwords with a key kept in special-purpose hardware, dividing information up amongst multiple servers [52] or one limited-bandwidth server [41]. Random passwords would also frustrate brute-force in this scenario, although the opportunity for key-stretching is probably closer to the 2 ms (14 bit) range to limit load on the login server.

3 Design

Given our estimation that a 56-bit secret can provide acceptable security against feasible brute-force attacks given a strong hash function and reasonable key stretching, our goal was to design a simple prototype interface that could train users to learn 56 bits secret with as little burden as possible.

Spaced repetition [43, 70, 62] typically employs delays (spacings) of increasing length between rehearsals of the chunk of information to be memorized. While precisely controlling rehearsal spacing makes sense in applications where education is users' primary goal, we did not want to interrupt users from their work. Instead, we chose to piggyback learning on top of an already-existing interruption in users' work-flow—the login process itself. We allow users to employ a user-chosen password for login, then piggyback learning of our assigned secret at the end of the login step. We split the 56-bit secret up into three equal-sized chunks to be learned sequentially, to enable a gradual presentation and make it as easy as possible for users to start typing from memory.

3.1 Encoding the secret

Although previous studies have found no significant differences in user's ability to memorize a secret encoded as words or letters [77, 64], we implemented both encodings. For *letters*, we used a string of 12 lowercase letters chosen uniformly at random from the English alphabet to encode a $26^{12} \approx 56.4$ bit secret. The three chunks of the secret were 4 letters each (representing ≈ 18.8 bits each).

For *words*, we chose a sequence of 6 short, common English words. To keep security identical to that of the *letters* case, we created our own list of 676 (26^2) possible words such that 6 words chosen uniformly at random would encode a $676^6 = 26^{12} \approx 56.4$ bit secret. We extracted all 3–5 English nouns, verbs and adjectives (which users tend to prefer in passwords [24, 85]) from Wiktionary, excluding those marked as vulgar or slang words and plural nouns. We also manually filtered out potentially insulting or negative words. From these candidate words we then greedily built our dictionary of 676 words by repeatedly choosing the most common remaining word, ranked by frequency in the Google N-gram

web corpus [27]. After choosing each word we then removed all words within an edit distance of two from the remaining set of candidates to potentially allow limited typo correction. We also excluded words which were a complete prefix of any other word, to potentially allow auto-complete. We present the complete list in Table 3.

3.2 Login form and hinting

Unlike typical login forms, we do not present a button to complete sign-in, but rather automatically submit the password for verification via AJAX each time a character is typed. Above the password field we display the word "verifying" while awaiting a response and "not yet correct" while the current text is not the correct password.

After the user's self-chosen password is verified, a text box for entering the first chunk of the user's assigned code appears to the right of the password field, as we show in Figure 1. On the first login, we display the correct value of the chunk immediately above the field into which users must enter it. In the version used for our study, we included a pop-up introducing the security code and its purpose:

> Due to concerns about stolen accounts and bonuses, we are giving you an additional security code. To finish logging in, simply type the [four letters | two words] above the text box. Your code will not change, so once you have learned it, try to type it before the hint appears.

We color each character a user enters into the security code field green if it is correct and red if incorrect. We replace correct characters with a green circle after 250 ms.

With each consecutive login, we delay the appearance of the hint by $\frac{1}{3}$ of a second for each time the user has previously seen the chunk, up to a maximum of 10 seconds. If the user types a character correctly before the delay expires, we start the delay countdown again. We selected these delay values with the goal of imposing the minimal annoyance necessary to nudge users to start typing from memory.

After a user enters a chunk without seeing the hint on three consecutive logins, we add another chunk. In the version used in our study, we show a pop-up which can be dismissed for all future logins:

> Congratulations! You have learned the first [four letters | two words] of your security code. We have added another [four letters | two words]. Just like the first [four letters | two words], once you have learned them, you can type them without waiting for the hint to appear.

When we detect that a user has finished typing the first chunk of their security code, we automatically tab (moved the cursor) to the text field for the second chunk and then start the delay for that chunk's hint. After typing the second chunk correctly from memory three times

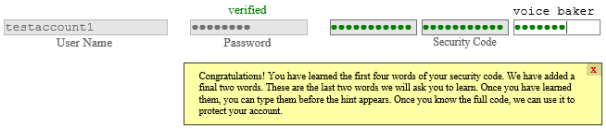


Figure 2: The login form for a user in who has just received the third security code chunk *words*.

in a row, we add the third and final chunk. In the version used in the study, we also displayed one more pop-up:

> Congratulations! You have learned the first [eight letters | four words] of your security code. We have added a final [four letters | two words]. These are the last [four letters | two words] we will ask you to learn. Once you have learned them, you can type them before the hint appears. Once you know the full code, we can use it to protect your account.

We illustrate the login process from our study, using all three chunks, in Figure 2. In a real deployment, once the user is consistently typing the entire security code from memory, entering their self-chosen password would no longer be necessary.

We disable pasting and automatic form-filling for the security code field to encourage users to type from memory. We allow users to type their code in lower or upper case, with all non-letter characters being ignored, including spaces between words as no word is a prefix of any other word. During training we automatically insert a space at the end of any entered code word so users learn that they do not need to type the spaces.

4 Experimental Methodology

We used a remote online study to evaluate our system. To keep participants from realizing the purpose of our study was the security codes and potentially altering their behavior, we presented our study as a psychology study with the security codes a routine part of logging in to participate. We recruited participants using Amazon's Mechanical Turk (MTurk) platform [59] and paid them to participate, which required logging in 90 times in 15 days. For completeness, we provide exact study materials the extended version of this paper [23].

4.1 The distractor task

We intended our distractor task to provide a plausible object of study that would lead us to ask participants to log in to our website repeatedly (distracting participants from the subject of our investigation) and to require a non-trivial mental effort (distracting them from making conscious efforts to memorize their security codes). Yet we also wanted the distractor task to be relatively fast,

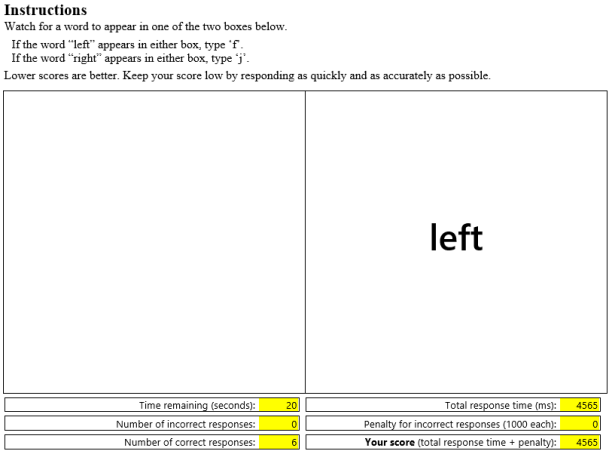


Figure 3: The Attention Game, our distractor task

interesting, and challenging, since we were asking participants to perform a large number of logins.

We designed a game to resemble descendants of the classic psychological study that revealed the Stroop effect [79]. Our game measured participants' ability to ignore where a word appeared (the left or right side of their screen) and respond to the meaning of the word itself. Each 60-second game consisted of 10 trials during which either the word 'left' or 'right' would appear in one of two squares on the screen, as illustrated in Figure 3. The words appeared in a random square after a random delay of 2–4 seconds, after which participants were asked to immediately press the `f` key upon seeing the word 'left' or `j` key upon seeing the word 'right' (corresponding to the left and right sides of a QWERTY keyboard). During the game, participants saw a score based on their reaction time, with penalties for pressing the wrong key.

4.2 Treatments

We randomly assigned participants to three treatments: *letters* (40% of participants), *words* (40%), and *control* (20%). Participants in the *letters* and *words* treatments received security codes consisting of letters and words, respectively, as described in Section 3.1. Participants in the *control* treatment received no security code at all and saw a simple password form for all logins; we included this treatment primarily to gauge whether the additional security codes were causing participants to drop out of the experiment more than traditional authentication would have.

4.3 Recruiting

We recruited participants to our study using Amazon's Mechanical Turk by posting a Human Intelligence Task (HIT) titled "60-Second Attention Study", paying US$0.40, and requiring no login. When participants completed the game, we presented them with an offer to "Earn $19 by being part of our extended study" (a screenshot of the offer is in the extended version of this paper [23]). The offer stated that participants would be required to play the game again 90 times within 15 days, answer two short questions before playing the game, wait 30 minutes after each game before starting a new game session, and that they would have to login for each session. We warned participants that those who joined the extended study but did not complete it would not receive partial payment. Our study prominently listed Microsoft Research as the institution responsible for the study. As we did not want to place an undue burden on workers who were not interested in even reading our offer, we provided a link with a large boldface heading titled "Get paid now for your participation in this short experiment" allowing participants to be paid immediately without accepting, or even reading, our offer.

When workers who had performed the single-game HIT signed up to participate in our 90-game attention study, we presented them with a sign-up page displaying our approved consent form and asking them to choose a username and a password of at least six characters. For the 88 logins following signup (games 2–89), and for login to the final session (in which we did not show the game but instead showed the final survey), we required participants to login using the chosen password and security code (if assigned a non-*control* treatment).

Amazon's policies forbid HITs that require workers to sign up for accounts on websites or to provide their email addresses. These rules prevent a number of abusive uses of Mechanical Turk. They also protect Amazon's business by forbidding requesters from recruiting workers, establishing a means of contact that bypasses Amazon, and then paying hired workers for future tasks without paying Amazon for its role in recruiting the workers. Our HIT was compliant with the letter of these rules because we only required workers to play the attention game, and they were in no way obligated to sign up for the full attention study. We were also compliant with the spirit of the rules, as we were not asking workers to engage in abusive actions and we did not cut Amazon out of their role as market maker—we paid participants for the 90-game attention study by posting a bonus for the HIT they already completed through Amazon.

As in any two-week project, some participants requested extensions to the completion deadline in order to reach 90 completed game. We provided 24-hour extensions to participants who were within 20 games of completing the study at the deadline.

How long has it been since you last slept for at least one hour without interruption?

Please indicate if you have consumed any of the following within the last 60 minutes:

- ☐ Food
- ☐ Beverages
- ☐ Caffeinated substances such as coffee or soft drinks
- ☐ Energy drinks other than caffeine

Figure 4: Participants were asked to fill out this two-question survey before every attention game.

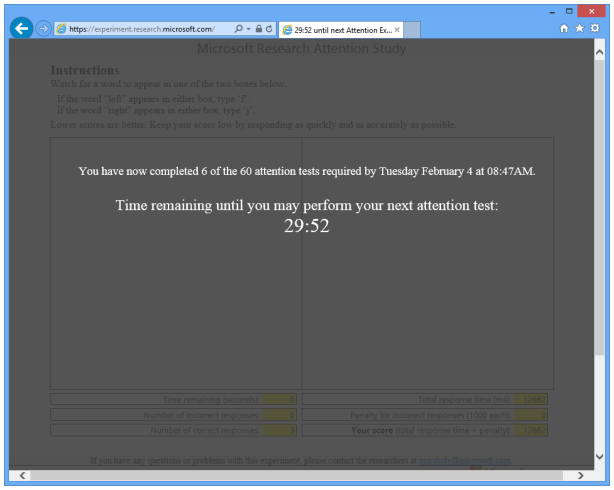


Figure 5: After completing an attention test, participants could not perform another one for 30 minutes.

4.4 Sessions

After each login we presented a very short survey (shown in Figure 4) asking participants about their recent sleep and eating. This was designed solely to support the purported goal of the study and we ignored the responses.

After participants completed the survey we immediately displayed the "Attention Game". When they completed the game, we overlaid a timer on top of the page counting down 30 minutes until they could again fill out the survey and play the game (see Figure 5). The timer also counted down in the title of the page, so that participants would see the countdown when browsing in other tabs and know when they were next allowed to play. If participants tried to log into the website again before the 30-minute waiting period was complete, we displayed the countdown again, starting from the amount of time remaining since they last completed the game.

4.5 Completion survey

When participants logged in for the 90th and final time, we skipped the game and displayed our completion survey. We provide the full text of the survey (with participants' answer counts) in the extended version of this paper [23]. We started the survey with demographic questions and then asked participants if they had written down or stored their passwords or assigned security codes outside of their memory.

We then debriefed participants about the true nature of the study, explaining that the security code was the focus of the study, though we did not reveal that we planned a follow-up study. We could not defer the debriefing to the follow-up study, as participants had not committed to engage with us beyond the end of the study and might not accept invitations for future contact. Indeed, some participants reported discussing the study in forums, but as we had entrusted those who finished the study with the truth, they returned that trust by respecting forum rules against 'spoilers' in all cases we are aware of.

To aid with the follow-up study, we asked participants to provide their email address, stating the question in a manner that we hoped would minimize suspicion that a formal follow-up study was imminent.

> If our analysis raises more questions about your experience during the study, may we contact you and ask you to answer a few questions in return for an additional bonus? If so, provide your email below. (This is optional!)

4.6 Payment

We paid $20 to participants who completed the study, as opposed to the $19 promised, to show extra gratitude for their participation. We informed participants of this only *after* they had completed the 'attention' study and filled out their post-deception ethics questionnaire, so as to not taint their responses about the ethics of the deception. However, this payment came well before the invitation to the follow-up study. Receiving a payment exceeding what we had promised may have increased participants' receptiveness to that invitation.

Despite telling participants they would not be paid unless they completed the study, we did pay $0.20 per login to all participants who logged into the site at least once after signing up. We did so because we couldn't be certain that the extra work of entering a security code didn't cause some participants to drop out. We wanted to ensure that if participants found the security code so arduous as to quit, they would not lose out on payment for the attention tests they did complete. We did not reveal this fact to the participants who completed the study and filled out the ethics survey as we feared they might communicate it to those who had yet to finish.

4.7 Follow-ups

At least 72 hours after a non-control group participant completed the study, we emailed them an invitation to perform an additional HIT for $1 (this email is reproduced in the extended version of this paper [23]). Most

participants provided an email address in the final survey of the attention study; we tried to contact those who didn't via Mechanical Turk. When participants accepted the HIT, we identified them by their Mechanical Turk ID to verify that they'd participated in the main study.[1]

The follow-up study contained only one question:

> Please try to recall and enter the security code that we assigned you during the attention study.
> **If you stored or wrote down your security code, please do not look it up.** We are only interested in knowing what you can recall using *only your memory*. It's OK if you don't remember some or all of it. Just do the best you can.

We presented participants with three text fields for the three chunks of their security code. Unlike the data-entry field used when they logged in for the attention experiment, we used plain text fields without any guidance as to whether the characters typed were correct. We accepted all responses from participants that arrived within two weeks of their completion of the study.

We emailed all participants who completed the first follow-up again 14 days after they completed it with the offer to complete a second identical follow-up for an additional $1 reward.

4.8 Ethics

The experiment was performed by Microsoft Research and was reviewed and approved by the organizations's ethics review process prior to the start of our first pilot.[2]

We employed deception to mask the focus of our research out of concern that participants might work harder to memorize a code if they knew it to be the focus of our study. We took a number of steps to minimize the potential for our deception to cause harm. We provided participants with estimates for the amount of time to complete the study padded to include the unanticipated time to enter the security code. While we told participants they would not be paid if they did not complete the study, we did make partial payments. We monitored how participants responded to the deception, investigating the responses of pilot participants before proceeding with the full study and continued to monitor participants in the full study, using a standard post-deception survey hosted by the Ethical Research Project [82]. We also offered participants the opportunity to withdraw their consent for use data derived from their participants. The vast majority of participants had no objection to the deception and none asked to have their data withdrawn. We provide more detail on participants' ethics responses in the extended version of this paper [23].

[1] We failed to verify that it had been three days since they completed the study, requiring us to disqualify three participants who discovered the follow-up study prematurely (see Section 5.1).

[2] The first author started a position at Princeton after the research was underway. He was not involved in the execution of the study or communications with participants. He did not have access to the email addresses of those participants who volunteered to provide them (the only personally-identifiable information collected).

5 Results

We present overall analysis of the most important results from our study: participant's ability to learn and recall security codes. We present a full accounting of participants' responses to the multiple-choice questions of our final survey and the complete text of that survey in the extended version of this paper [23], including demographics which reflect the typical MTurk population [74].

5.1 Recruitment and completion

We offered our initial attention-game task to roughly 300 workers from February 3–5, 2014. 251 workers accepted the offer to participate in our study by completing the sign-up page and playing the first game. We stopped inviting new participants when we had reached roughly 100 sign-ups for our two experimental groups. Participants' assigned treatment had no effect until they returned after sign-up and correctly entered their username and chosen password into the login page, so we discard the 28 who signed up but never returned. We categorize the 223 participants who did return in Table 1.

5.1.1 Dropouts

Inserting a security-code learning step into the login process creates an added burden for participants. Of participants who completed the study, typing (and waiting for) the security codes added a median delay of 6.9 s per login. To measure the impact of this burden, we tested the hypothesis that participants assigned a security code would be less likely to complete the experiment than those in the *control*. The null hypothesis is that group assignment has no impact on the rate of completion.

Indeed, the study-completion rates in the fourth row of Table 1 are higher for *control* than the experimental groups. We use a two-tailed Fisher's Exact Test to compare the proportion of participants who completed the study between those assigned a security code (the union of the *letters* and *words* treatments, or 133 of 170) to that of the *control* (35 of 41). The probability of this difference occurring by chance under the null hypothesis is $p = 0.2166$. While this is far from the threshold for statistical significance, such a test cannot be used to reject the alternate hypothesis that the observed difference reflects a real percentage of participants who dropped out due to the security code.

Digging into the data further, we can separate out those participants who abandoned the study after exactly

	Control		*Letters*		*Words*		*Total*	
Signed up for the 'attention' study	41		92		90		223	
Quit after 2 or 3 games	0/41	0%	9/92	10%	12/90	13%	21/223	9%
Otherwise failed to finish	6/41	15%	14/92	15%	12/90	13%	32/223	14%
Completed the 'attention' study	35/41	85%	69/92	75%	66/90	73%	170/223	76%
Received full security code	—		63/68	93%	64/65	98%	127/133	95%
Typed entire code from memory	—		62/63	99%	64/64	100%	126/127	99%
Participated in first follow-up	—		56/63	89%	56/64	88%	112/127	88%
Recalled code correctly	—		46/56	82%	52/56	93%	98/112	88%
Participated in second follow-up	—		52/56	93%	52/56	93%	104/112	93%
Recalled code correctly	—		29/52	56%	32/52	62%	61/104	59%

Table 1: Results summary: participants who signed up for the attention study, the fraction of those participants who completed the study, the fraction of the remaining participants who entered the first two chunks of their security code reliably enough to be shown the full security code (all three chunks), the fraction of those remaining who participated in the follow-up studies (after 3 and 17 days, respectively), and the fraction of those who recalled their security code correctly. The *control* group did not receive security codes and hence are excluded from the latter rows of the table.

two or three games from those who failed to finish later (no participant quit after the fourth or fifth games). While no participant in the *control* quit between two or three games, 9 participants assigned to *letters* and 12 assigned to *words* did. For participants who completed more than three games, the rate of failure to finish the study is remarkably consistent between groups. We do not perform statistical tests as this threshold is data-derived and any hypothesis based on it would be post-hoc. Rather, as our study otherwise presents a overall favorable view of random assigned secrets, we present the data in this way as it illustrates to the reader reason for skepticism regarding user acceptance among unmotivated participants.

5.1.2 Participants who appeared not to learn

Six participants completed the study without receiving all three chunks of their security codes, having failed to demonstrate learning by typing the first chunk (one participant from *letters*) or second chunk (five participants, four from *letters* and one from *words*) before the hint appeared. After the conclusion of the study we offered participants $1 to provide insights into what had happened and all replied. Two in the *letters* group, including the one who only received one chunk, reported genuine difficulty with memory. The other four stated quite explicitly (which we provide in the extended version of this paper [23]) that they purposely avoided revealing that they had learned the second chunk to avoid being assigned more to learn.

5.1.3 Excluded participants

We found it necessary to exclude four participants from some of our analysis. Three participants, two in *words* and one in *letters*, discovered and accepted the follow-up HIT before three days had passed since the end of the study, ignoring the admonition not to accept this HIT without an invitation. Though these participants all completed the 90-game attention study, learned and recalled their entire security code, we count them as having not returned for the follow-up. We corrected this bug prior to the second follow-up. We disqualified one additional 'participant' in the *letters* group which appeared to be using an automated script.

After revealing the deceptive nature of the study we gave participants the option to withdraw their consent for us to use our observations of their behavior, while still receiving full payment. Fortunately, none chose to do so.

5.2 Learning rates

Of non-*control* participants completing the study, 93% eventually learned their full security code well enough to type it from memory three times in a row (91% of *letters* and 96% of *words*). Most participants learned their security codes early in the study, after a median of 36 logins (37 for *letters* and 33 of *words*). We show the cumulative distribution of when participants memorized each chunk of their code in Figure 6.

We consider whether participants first typed their codes from memory in fewer logins with either *letters* or *words*, with the null hypothesis that encoding had no impact on this measure of learning speed. A two-tailed Mann-Whitney U (rank sum) test on the distribution of these two sets of learning speeds estimates a probability of $p = 0.07$ ($U = 1616$) of observing this difference by chance, preventing us from rejecting the null hypothesis.

We had hypothesized that, with each subsequent chunk we asked participants to memorize, their learn-

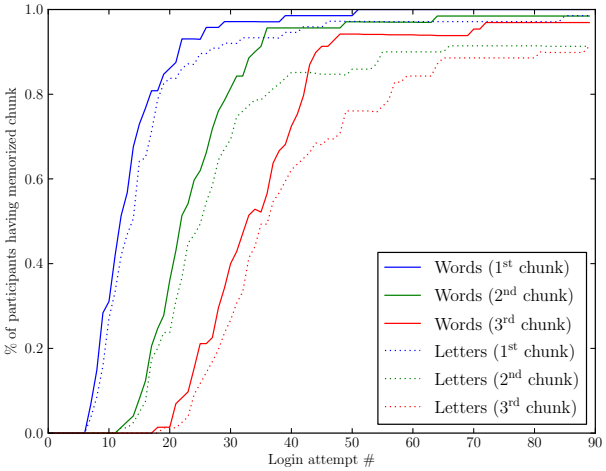


Figure 6: We show the proportion of participants who had memorized each chunk of their security code after a given number of login attempts. We considered a participant to have memorized a chunk after they entered it without a hint in three consecutive logins.

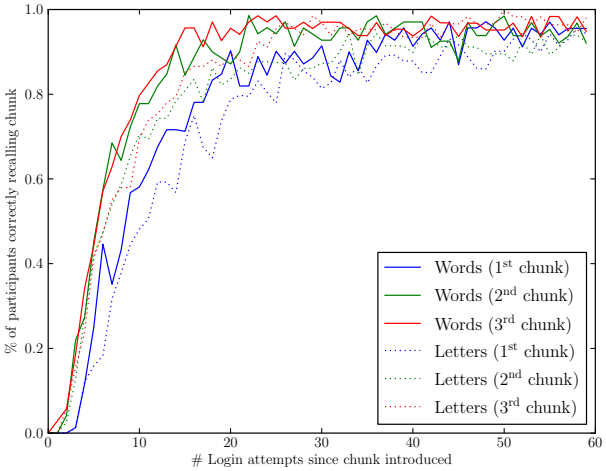


Figure 7: For each of the three chunks in participants' security codes, we show the proportion of participants who entered each chunk without a hint as a function of the number of previous exposures to the chunk (the number of previous logins in which the chunk appeared). On the whole, participants actually memorized their second and third chunks more quickly than the first.

ing speed might decrease due to interference [31] or increase due to familiarity with the system. Learning times decreased. We use a Mann-Whitney U test to compare learning times between the first and final chunks, using times only for participants who learned all three, yielding a significant $p < 0.001$ ($U = 4717$). To remove the impact of the time required to notice the delay and learn that they could enter the code before it appeared, we compare the learning times between the third and second chunks. This difference is much smaller, with a Mann-Whitney U test yielding a probability of $p = 0.39$ ($U = 7646$) of an effect due to chance.

To illustrate the increasing rate of learning we show, in Figure 7, the percent of participants who typed each chunk correctly from memory as a function of the number of previous exposures to that chunk.

5.3 Login speed and errors

Overall, participants in the *words* group took a median time of 7.7 s to enter their security codes, including waiting for any hints to appear that they needed, and participants in the *letters* group took a median time of 6.0 s. Restricting our analysis to those logins in which participants were required to enter all three chunks of the code only increases the median login time to 8.2 s for *words* and 6.1 s for *letters*.[3] The distribution had a relatively long tail, however, with the 95th percentile of logins taking 23.6 s for *words* and 20.5 s for *letters*.

[3]The median login time actually went down for *letters* participants when all three chunks were required, likely because this included more logins typed exclusively from memory with no waiting for a hint.

After computing the median login time for each participant, we compared the set of these values for participants in the two experimental groups using a Mann-Whitney U. We can reject the null hypothesis that the differences between these medians were the result of chance with $p < 0.01$ ($U = 1452$) and conclude that participants in the *letters* group were significantly faster.

Errors in entering security codes (whether typos or genuine memory errors) were relatively rare: over all 90 logins participants in the *words* group made fewer errors (with a median of 5) than participants in the *letters* group (median 7). Using a Mann-Whitney U, we cannot reject the null hypothesis that neither group would make more errors than the other ($p = 0.08$ ($U = 1706$)).

5.4 Recall of security codes in follow-ups

We sent invitations to participants to follow-up studies testing recall of their security codes 3 days after the initial study ended and then 14 more days after they completed the first follow-up. The median time between when participants completed the study and actually took the first follow-up study was 3 days 18 hours (mean 4 days 23 hours). For the second follow-up study the median time was 16 days 0 hours (mean 16 days 13 hours). By comparison, the median time to complete the study itself was 10 days 5 hours (mean 9 days 19 hours).

Overall, 88% of participants recalled their code correctly in the first follow-up and 59% did so in the second. The drop-off at the second follow-up was expected

as memory is believed to decay exponentially with the delay since the information was last recalled [89].

We had hypothesized that participants in the *letters* treatment might be more or less likely to recall their security codes correctly in the follow-ups than participants in the *words* treatment. As seen in Table 1, of participants in the *letters* group 82% recalled their security codes correctly in the first follow-up and 56% did so in the second study, compared to 93% and 62%, respectively, of users in *words*. Using a two-tailed Fisher's Exact Test, we cannot rule out the null hypothesis that participants in either group were equally likely to recall codes correctly, with the observed differences occurring with a $p = 0.15$ chance in the first follow-up and $p = 0.45$ in the second follow-up under the null hypothesis.

5.4.1 Types of errors

We observed 14 participants incorrectly entering their code in the first follow-up and 52 in the second. All 13 users who entered incorrectly in the first follow-up and participated in the second entered their code incorrectly again. This sample is too low to draw firm conclusions about the nature of participants' recall errors, but we did see evidence that users retained partial memory, with 75% of users entering at least one component of their code correctly in the second follow-up and 48% missing only one component or entering components in the wrong order. Re-arranging the order of components, which accounted for 10% of errors, could be corrected by accepting components in any order at a loss of only $\log_2(3!) \approx 2.6$ bits of security. Unfortunately, the majority of other errors could not be corrected without significantly downgrading security. Only 3 participants (6%) in the second-followup (and 2 in the first) entered a code within an edit distance of 2 of the correct code. We present further information on the types of errors observed in the extended version of this paper [23].

5.4.2 Storing security codes

A minority of participants reported storing their security code outside of their memory, as presented in Table 2. We were concerned that participants who had stored their security codes might have been tempted to look them up and thereby inflated the recall rate during the follow-up. However, only 82% of participants storing their security code recalled it correctly on follow-up, whereas 89% of participants not storing the security code did. While it's possible that participants who did not rely on a stored code were better able to remember as a result, we had not hypothesized this in advance nor would the differences we observed have been statistically significant.

We had hypothesized that participants might be more likely to write down or otherwise store codes outside their memory if assigned a code composed of letters as opposed to words, or vice versa. The null hypothesis is that treatment has no impact on the choice to store codes. In the completion survey, 18 of the 69 participants in the *letters* treatment reported having stored their security code, as compared to 10 of the 66 in the *words* treatment. We use a two-sided Fisher's Exact Test to estimate that such a difference would occur with probability $p = 0.14$ under the null hypothesis. Thus we can not conclude that either treatment made participants more likely to write their code down.

6 Limitations

Whenever testing a new approach to security, its novelty alone may be enough to reveal to research participants that it is the focus of the study. Despite our best efforts, of the 133 participants in the experimental groups who completed the study (68 in *letters* and 65 in *words*), only 35 (26%, 24 from *letters* and 11 from *words*) reported that they did not suspect that the security code might be the focus of the study. The majority, 70 (53%, 28 from *letters* and 42 from *words*) reported having some suspicion and 28 (21%, 16 from *letters* and 12 from *words*) reported being 'certain' the security code was the focus of the study. Still, to our knowledge no participants revealed any 'spoilers' on public forums. Participants who suspected we were studying their ability to learn the security code may have tried harder to memorize the code than if they had not, though it's not clear how their effort would compare to that of a real-world user relying on a randomly-assigned code to secure something valuable.

7 Background and related work

7.1 Physiological principles of memory

Human memory has been studied extensively by psychologists (as well as neuroscientists and others). The *spacing effect* describes how people are better able to recall information if it is presented for the same duration, but in intervals spaced over a longer period of time. This effect was first described in the 19$^{\text{th}}$ century [43] and is considered one of the most robust memory effects [10]. It has even been demonstrated in animals. The effect is almost always far more powerful than variations in memory between individual people [35].

The cause of the spacing effect is still under debate, but most theories are based around the *multi-store model* of memory [33] in which short-term (or working memory) and long-term memory are distinct neurological processes [8, 9]. One theory of the spacing effect posits

	Did you store any part of the additional security code for the study website, such as by writing it down, emailing it to yourself, or adding it to a password manager? 'Yes'				'No'			
	Letters		*Words*		*Letters*		*Words*	
Completed the study	18/68	26%	10/65	15%	50/68	74%	55/65	85%
Reported storing password	11/18	61%	6/10	60%	2/50	4%	0/55	0%
Received full security code	16/18	89%	9/10	90%	47/50	94%	55/55	100%
Participated in follow-up	14/16	88%	8/9	89%	42/47	89%	48/55	87%
Recalled code correctly	12/14	86%	6/8	75%	34/42	81%	46/48	96%

Table 2: A minority of participants reported storing their security code outside of their memory. Each row corresponds to an identically-named row in Table 1, separated by participants' response to the code storage question in each column. The first row shows the fraction of all participants who completed the study in each group, and each subsequent row as a fraction of the one above, except for the italicized row which identifies participants who reported storing their self-chosen password (which was much more common amongst participants who stored their security code).

that when information is presented which has left short-term memory, a trace of it is recognized from long-term memory [47] and hence stimulated, strengthening the long-term memory through *long-term potentiation* [14] of neural synapses. Thus, massed presentation of information is less effective at forming long-term memories because the information is recognized from working memory as it is presented. In our case, the natural spacing between password logins is almost certainly long enough for the password to have left working memory.

Early work on spaced learning focused on *expanding presentation* in which an exponentially increasing interval between presentations was considered optimal [70, 62]. More recent reviews have suggested that the precise spacing between presentations is not particularly important [11] or that even spacing may actually be superior [53]. This is fortunate for our purposes as password logins are likely to be relatively evenly spaced in time. Other work has focused on dynamically changing spacing using feedback from the learner such as speed and accuracy of recall [68] which could potentially guide artificial rehearsal of passwords.

7.2 Approaches to random passwords

Many proposals have aimed to produce random passwords which are easier for humans to memorize, implicitly invoking several principles of human memory. Early proposals typically focused on *pronounceable* random passwords [46, 90] in which strings were produced randomly but with an English-like distribution of letters or phonemes. This was the basis for NIST's APG standard [2], though that specific scheme was later shown to be weak [45]. The independently-designed `pwgen` command for generating pronounceable passwords is still distributed with many Unix systems [5].

Generating a random set of words from a dictionary, as we did in our *words* treatment, is also a classic approach, now immortalized by the web comic XKCD [67]. This was first proposed by Kurzban [61] with a very small 100 word dictionary, the popular Diceware project [6] offers 4,000 word dictionaries. Little available research exists on what size and composition of dictionaries is optimal.

Finally, a number of proposals have aimed to enhance memorability of a random string by offering a secondary coding such as a set of images [58], a grammatical sentence [7, 50], or a song [65]. Brown's passmaze protocol was recognition-based, with users simply recognizing words in a grid [29]. None of these proposals has received extensive published usability studies.

7.3 Studies on password recall

A number of studies have examined user performance in recalling passwords under various conditions. These studies often involve users choosing or being assigned a new password in an explicitly experimental setting, and testing the percentage of users who can correctly recall their password later. Surprisingly, a large number of studies have failed to find any statistically significant impact on users' ability to recall passwords chosen under a variety of experimental treatments, including varying length and composition requirements [95, 71, 92, 86, 60] or requiring sentence-length passphrases [55].[4] The consistent lack of impact of password structure on recall rates across studies appears to have gone unremarked in any of the individual studies.

However, several studies have found that stricter composition requirements increase the number of users writing their passwords down [71, 60] and users self-report that they believe passwords are harder to remember when created under stricter password policies [60, 92].

[4] Keith et al. [55] did observe far more typos with sentence-length passwords, which needed correcting to isolate the effective recall rates.

At least three studies have concluded that users *are* more likely to remember passwords they use with greater frequency [95, 28, 42]. This suggests that lack of adequate training may in fact be the main bottleneck to password memorization, rather than the inherent complexity of passwords themselves. Brostoff [28] appears to have made the only study of password *automacity* (the ability to enter a password without consciously thinking about it), and estimated that for most users, this property emerges for passwords they type at least once per day.

A few studies have directly compared recall rates of user-generated passwords to assigned passwords. Interestingly, none has been able to conclude that users were less likely to remember assigned passwords. For example, in a 1990 study by Zviran and Haga [94] in which users were asked to generate a password and then recall it 3 months later, recall was below 50% for all unprompted text passwords and no worse for system-assigned random passwords, though the rate of writing increased. A similar lab study by Bunnell et al. found a negligibly smaller difference in recall rate for random passwords [30]. A 2000 study by Yan et al. [92] found that users assigned random passwords for real, frequently-used accounts actually requested fewer password resets than users choosing their own passwords, though those users were also encouraged to write their passwords down "until they had memorized them." Stobert in 2011 [78] found no significant difference in recall between assigned and user-chosen text passwords.

Two studies have exclusively compared user's ability to recall random passwords under different encodings. The results of both were inconclusive, with no significant difference in recall rate between users given random alphanumeric strings, random pronounceable strings or randomly generated passphrases at a comparable security level of 30 [77] or 38 bits [64]. The results appear robust to significant changes in the word dictionary used for passwords or the letters used in random strings. However, users stated that alphanumeric strings seemed harder to memorize than random passphrases [77].

All of these studies except that of Yan et al. face validity concerns as the passwords were explicitly created for a study of password security. A 2013 study by Fahl et al. [44] compared user behavior in such scenarios and found that a non-trivial proportion of users behave significantly differently in explicit password studies by choosing deliberately weak passwords, while a large number of users re-use real passwords in laboratory studies. Both behaviors bias text passwords to appear more memorable, as deliberately weak passwords may be easy to memorize and existing passwords may already be memorized. Also of concern, all of these studies (again excluding Yan et al.) involved a single enrollment process followed by recall test, with no opportunity for learning.

Spaced repetition for passwords was recently suggested by Blocki et al. [16], who proposed designing password schemes which insert a minimal number of artificial rehearsals to maintain security. After our study, Blocki published results from a preliminary study on mnemonic passwords with formal rehearsals [15]. Compared to our study, participants performed a much lower number of rehearsals spaced (about 10) spaced over a longer period (up to 64 days), prompted by the system at specific times rather than at the participant's convenience. Unlike our study participants were aware that memorization was the explicit goal of the study. Blocki also incorporated additional mnemonic techniques (images and stories). This study provides evidence that spaced repetition and other techniques can be applied more aggressively for motivated users, whereas as our study demonstrates the practicality with few system changes and unmotivated users.

7.4 Alternative authentication schemes

Several approaches have been explored for exploiting properties of human memory in authentication systems. One approach is to query already-held memories using *personal knowledge question* schemes such as "what is your mother's maiden name?" though more sophisticated schemes have been proposed [93, 48] While these schemes typically enable better recall than passwords, they are vulnerable to attacks by close social relations [76], many people's answers are available in online search engines or social networks [73], and many questions are vulnerable to statistical guessing [21, 76]. An advantage of personal knowledge questions is that they represent *cued recall* with the question acting as a cue, which generally increases memory performance over *free recall*.

Graphical passwords aim to utilize humans' strong abilities to recognize visual data [13]. Some schemes employ cued recall only by asking users to recognize a secret image from a set [40, 87, 80]. Others use uncued memory by asking users to draw a secret pattern [49, 81, 12] or click a set of secret points in an image [88, 37]. These schemes are often still vulnerable to guessing attacks due to predictable user choices [39, 83, 84]. The Persuasive Cued Click-Points scheme [36] attempts to address this by forcing users to choose points within a system-assigned region, which was not found to significantly reduce recall. Still, it remains unclear exactly what level of security is provided by most graphical schemes and they generally take longer to authentication than typing a text password. They have found an important niche on mobile devices with touch screens, with current versions of Android and Windows 8 deploying graphical schemes for screen unlock.

Bojinov et al. [17] proposed the use of implicit memory for authentication, training users to type a random key sequence in rapid order using a game similar to one used in psychological research to study implicit memory formation [75]. After a 30–45 minute training period, users were tested 1–2 weeks later on the same game with their trained sequences and random sequences, with about half performing significantly better on trained sequences. Such a scheme offers the unique property that users are unaware of their secret and thus incapable of leaking it to an attacker who doesn't know the correct secret challenge to test on, providing a measure of resistance against "rubber-hose" attacks (physical torture). Without dramatic improvements however this scheme is impractical for normal personal or corporate logins due to the very long enrollment and login times and the low rate of successful authentication.

8 Open questions and future work

As this was our first exploration of spaced repetition for learning random secrets, many of our design choices were best guesses worthy of further exploration. The character set used when encoding secrets as letters, namely 26 lowercase letters, might be inferior to an expanded set such as base-32 with digits included [51]. Our choice of a dictionary of 676 words is almost surely not optimal, since we deliberately chose it for equivalence to the size of our character set. Splitting the secret into three equal-sized chunks was also simply a design heuristic, performance might be better with more or fewer chunks.

We expect spaced repetition to be a powerful enough tool for users to memorize secrets under a variety represenation formats, though the precise details may have important implications. We observed letters to be slightly faster to type and words slightly faster to learn. We also observed double the rate of forgotten codes after three days in the *letters* group and, though this difference was not statistically significant given our sample sizes and the low absolute difference, this is worthy of further study as this difference could be important in practice.

Our system can likely be improved by exploiting additional memory effects, such as dual-coding secrets by showing pictures next to each word or requiring greater depth of processing during each rehearsal. Cued recall could also be utilized by showing users random prompts (images or text) in addition to a random password.

On the downside, interference effects may be a major hindrance if users were asked to memorize multiple random passwords using a system like ours. This is worthy of further study, but suggests that random passwords should only be used for critical accounts.

Changing the login frequency may decrease or increase performance. We aimed to approximate the number of daily logins required in an enterprise environment in which users lock their screen whenever leaving their desks. In this context, the trade-offs appear reasonable if newly-enrolled users can learn a strong password after two weeks of reduced security (to the level of a user-chosen password) with about 10 minutes of aggregate time spent learning during the training period.

In contexts with far fewer logins, such as password managers or private keys which might be used once per day or less, learning might require a larger number of total logins. If a higher total number of logins are needed and they occur at a slower rate, this may lead to an unacceptable period of reduced security. In this case, security-conscious users could use rehearsals outside of authentication events. Further, if codes are used extremely infrequently after being memorized, artificial rehearsals may be desirable even after learning the secret. These are important cases to study, in particular as these are cases in which there is no good alternative defense against offline brute-force attacks.

While the learning rates of our participants did not slow down as the number of chunks they memorized increased, they might have more have trouble as the number of chunks grows further or as they have to associate different codes with different accounts. Fortunately, most users only have a small number of accounts valuable enough to require a strong random secret.

9 Conclusion

For those discouraged by the ample literature detailing the problems that can result when users and security mechanisms collide, we see hope for the human race. Most users *can* memorize strong cryptographic secrets when, using systems freed from the constraints of traditional one-time enrollment interfaces, they have the opportunity to learn over time. Our prototype system and evaluation demonstrate the brain's remarkable ability to learn and later recall random strings—a fact that surprised even participants at the conclusion of our study.

10 Acknowledgments

We thank Janice Tsai for assistance and suggestions on running an ethical experiment, Serge Egelman and David Molnar for help running our experiment on Mechanical Turk, and Arvind Narayanan, Cormac Herley, Paul van Oorschot, Bill Bolosky, Ross Anderson, Cristian Bravo-Lilo, Craig Agricola and our anonymous peer reviewers for many helpful suggestions in presenting our results.

References

[1] HashCat project. `http://hashcat.net/hashcat/`.

[2] "Automated Password Generator (APG)". *NIST Federal Information Processing Standards Publication* (1993).

[3] Bitcoin currency statistics. `blockchain.info/stats`, 2014.

[4] ADAMS, A., SASSE, M. A., AND LUNT, P. Making passwords secure and usable. In *People and Computers XII*. Springer London, 1997, pp. 1–19.

[5] ALLBERY, B. pwgen—random but pronounceable password generator. *USENET posting in comp.sources.misc* (1988).

[6] ARNOLD, R. G. The Diceware Passphrase Home Page. `world.std.com/~reinhold/diceware.html`, 2014.

[7] ATALLAH, M. J., MCDONOUGH, C. J., RASKIN, V., AND NIRENBURG, S. Natural language processing for information assurance and security: an overview and implementations. In *Proceedings of the 2000 New Security Paradigms Workshop* (2001), ACM, pp. 51–65.

[8] ATKINSON, R. C., AND SHIFFRIN, R. M. Human memory: A proposed system and its control processes. *The Psychology of Learning and Motivation 2* (1968), 89–195.

[9] BADDELEY, A. Working memory. *Science 255*, 5044 (1992), 556–559.

[10] BADDELEY, A. D. *Human memory: Theory and practice*. Psychology Press, 1997.

[11] BALOTA, D. A., DUCHEK, J. M., AND LOGAN, J. M. Is expanded retrieval practice a superior form of spaced retrieval? A critical review of the extant literature. *The foundations of remembering: Essays in honor of Henry L. Roediger, III* (2007), 83–105.

[12] BICAKCI, K., AND VAN OORSCHOT, P. C. A multi-word password proposal (gridWord) and exploring questions about science in security research and usable security evaluation. In *Proceedings of the 2011 New Security Paradigms Workshop* (2011), ACM, pp. 25–36.

[13] BIDDLE, R., CHIASSON, S., AND VAN OORSCHOT, P. C. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys (CSUR) 44*, 4 (2012), 19.

[14] BLISS, T. V., AND LØMO, T. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of Physiology 232*, 2 (1973), 331–356.

[15] BLOCKI, J. *Usable Human Authentication: A Quantitative Treatment*. PhD thesis, Carnegie Mellon University, June 2014.

[16] BLOCKI, J., BLUM, M., AND DATTA, A. Naturally rehearsing passwords. In *Advances in Cryptology-ASIACRYPT 2013*. Springer, 2013, pp. 361–380.

[17] BOJINOV, H., SANCHEZ, D., REBER, P., BONEH, D., AND LINCOLN, P. Neuroscience meets cryptography: designing crypto primitives secure against rubber hose attacks. In *Proceedings of the 21st USENIX Security Symposium* (2012).

[18] BONNEAU, J. *Guessing human-chosen secrets*. PhD thesis, University of Cambridge, May 2012.

[19] BONNEAU, J. Moore's Law won't kill passwords. Light Blue Touchpaper, January 2013.

[20] BONNEAU, J., HERLEY, C., VAN OORSCHOT, P. C., AND STAJANO, F. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *2012 IEEE Symposium on Security and Privacy* (May 2012).

[21] BONNEAU, J., JUST, M., AND MATTHEWS, G. What's in a Name? Evaluating Statistical Attacks on Personal Knowledge Questions. In *FC '10: Proceedings of the the 14th International Conference on Financial Cryptography* (January 2010).

[22] BONNEAU, J., PREIBUSCH, S., AND ANDERSON, R. A birthday present every eleven wallets? The security of customer-chosen banking PINs. In *FC '12: Proceedings of the the 16th International Conference on Financial Cryptography* (March 2012).

[23] BONNEAU, J., AND SCHECHTER, S. Towards reliable storage of 56-bit secrets in human memory (extended version). Tech. rep., Microsoft Research.

[24] BONNEAU, J., AND SHUTOVA, E. Linguistic properties of multi-word passphrases. In *USEC '12: Workshop on Usable Security* (March 2012).

[25] BOYEN, X. Halting password puzzles. In *USENIX Security Symposium* (2007).

[26] BRAND, S. Department of Defense Password Management Guideline.

[27] BRANTZ, T., AND FRANZ, A. The Google Web 1T 5-gram corpus. Tech. Rep. LDC2006T13, Linguistic Data Consortium, 2006.

[28] BROSTOFF, A. *Improving password system effectiveness*. PhD thesis, University College London, 2004.

[29] BROWN, D. R. Prompted User Retrieval of Secret Entropy: The Passmaze Protocol. *IACR Cryptology ePrint Archive 2005* (2005), 434.

[30] BUNNELL, J., PODD, J., HENDERSON, R., NAPIER, R., AND KENNEDY-MOFFAT, J. Cognitive, associative and conventional passwords: Recall and guessing rates. *Computers & Security 16*, 7 (1997), 629–641.

[31] BUNTING, M. Proactive interference and item similarity in working memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition 32*, 2 (2006), 183.

[32] BURR, W. E., DODSON, D. F., AND POLK, W. T. Electronic Authentication Guideline. *NIST Special Publication 800-63* (2006).

[33] CAMERON, K. A., HAARMANN, H. J., GRAFMAN, J., AND RUCHKIN, D. S. Long-term memory is the representational basis for semantic verbal short-term memory. *Psychophysiology 42*, 6 (2005), 643–653.

[34] CAPLE, C. *The Effects of Spaced Practice and Spaced Review on Recall and Retention Using Computer Assisted Instruction.* PhD thesis, North Carolina State University, 1996.

[35] CEPEDA, N. J., PASHLER, H., VUL, E., WIXTED, J. T., AND ROHRER, D. Distributed practice in verbal recall tasks: A review and quantitative synthesis. *Psychological Bulletin 132*, 3 (2006), 354.

[36] CHIASSON, S., FORGET, A., BIDDLE, R., AND VAN OORSCHOT, P. C. Influencing users towards better passwords: persuasive cued click-points. In *Proceedings of the 22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction-Volume 1* (2008), British Computer Society, pp. 121–130.

[37] CHIASSON, S., VAN OORSCHOT, P. C., AND BIDDLE, R. Graphical password authentication using cued click points. In *Computer Security–ESORICS 2007*. Springer, 2007, pp. 359–374.

[38] CLAIR, L. S., JOHANSEN, L., ENCK, W., PIRRETTI, M., TRAYNOR, P., MCDANIEL, P., AND JAEGER, T. Password exhaustion: Predicting the end of password usefulness. In *Information Systems Security*. Springer, 2006, pp. 37–55.

[39] DAVIS, D., MONROSE, F., AND REITER, M. K. On User Choice in Graphical Password Schemes. In *USENIX Security Symposium* (2004), vol. 13, pp. 11–11.

[40] DHAMIJA, R., AND PERRIG, A. Déjà Vu: A User Study Using Images for Authentication. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9* (Berkeley, CA, USA, 2000), SSYM'00, USENIX Association, pp. 4–4.

[41] DI CRESCENZO, G., LIPTON, R., AND WALFISH, S. Perfectly secure password protocols in the bounded retrieval model. In *Theory of Cryptography*. Springer, 2006, pp. 225–244.

[42] DUGGAN, G. B., JOHNSON, H., AND GRAWEMEYER, B. Rational security: Modelling everyday password use. *International Journal of Human-Computer Studies 70*, 6 (2012), 415–431.

[43] EBBINGHAUS, H. *Über das gedächtnis: untersuchungen zur experimentellen psychologie*. Duncker & Humblot, 1885.

[44] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security* (2013), ACM, p. 13.

[45] GANESAN, R., DAVIES, C., AND ATLANTIC, B. A new attack on random pronounceable password generators. In *Proceedings of the 17th {NIST}-{NCSC} National Computer Security Conference* (1994).

[46] GASSER, M. A random word generator for pronounceable passwords. Tech. rep., DTIC Document, 1975.

[47] GREENE, R. L. Spacing effects in memory: Evidence for a two-process account. *Journal of Experimental Psychology: Learning, Memory, and Cognition 15*, 3 (1989), 371.

[48] JAKOBSSON, M., YANG, L., AND WETZEL, S. Quantifying the security of preference-based authentication. In *Proceedings of the 4th ACM Workshop on Digital Identity Management* (2008), ACM, pp. 61–70.

[49] JERMYN, I., MAYER, A., MONROSE, F., REITER, M. K., RUBIN, A. D., ET AL. The design and analysis of graphical passwords. In *Proceedings of the 8th USENIX Security Symposium* (1999), vol. 8, Washington DC, pp. 1–1.

[50] JEYARAMAN, S., AND TOPKARA, U. Have the cake and eat it too—Infusing usability into text-password based authentication systems. In *Computer Security Applications Conference, 21st Annual* (2005), IEEE.

[51] JOSEFSSON, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006.

[52] JUELS, A., AND RIVEST, R. L. Honeywords: Making Password-cracking Detectable. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 145–160.

[53] KARPICKE, J. D., AND ROEDIGER III, H. L. Expanding retrieval practice promotes short-term retention, but equally spaced retrieval enhances long-term retention. *Journal of Experimental Psychology: Learning, Memory, and Cognition 33*, 4 (2007), 704.

[54] KAUFMAN, C., PERLMAN, R., AND SPECINER, M. *Network security: Private communication in a public world*. Prentice Hall Press, 2002.

[55] KEITH, M., SHAO, B., AND STEINBART, P. J. The usability of passphrases for authentication: An empirical field study. *International Journal of Human-Computer Studies 65*, 1 (2007), 17–28.

[56] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 523–537.

[57] KELSEY, J., SCHNEIER, B., HALL, C., AND WAGNER, D. Secure applications of low-entropy keys. In *Information Security*. Springer, 1998, pp. 121–134.

[58] KING, M. Rebus passwords. In *Proceedings of the Seventh Annual Computer Security Applications Conference, 1991* (Dec 1991), pp. 239–243.

[59] KITTUR, A., CHI, E. H., AND SUH, B. Crowdsourcing User Studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2008), CHI '08, ACM, pp. 453–456.

[60] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), ACM, pp. 2595–2604.

[61] KURZBAN, S. A. Easily Remembered Passphrases: A Better Approach. *SIGSAC Rev. 3*, 2-4 (Sept. 1985), 10–21.

[62] LANDAUER, T., AND BJORK, R. Optimum rehearsal patterns and name learning. InM. M. Gruneberg, PE Morris, & RN Sykes (Eds.), Practical aspects of memory (pp. 625-632), 1978.

[63] LASTPASS. LastPass Security Notification. http://blog.lastpass.com/2011/05/lastpass-security-notification.html.

[64] LEONHARD, M. D., AND VENKATAKRISHNAN, V. A comparative study of three random password generators. In *IEEE EIT* (2007).

[65] MEUNIER, P. C. Sing-a-Password: Quality Random Password Generation with Mnemonics. 1998.

[66] MORRIS, R., AND THOMPSON, K. Password Security: A Case History. *Communications of the ACM 22*, 11 (1979), 594–597.

[67] MUNROE, R. Password Strength. `https://www.xkcd.com/936/`, 2012.

[68] PAVLIK, P. I., AND ANDERSON, J. R. Using a model to compute the optimal schedule of practice. *Journal of Experimental Psychology: Applied 14*, 2 (2008), 101.

[69] PERCIVAL, C. Stronger key derivation via sequential memory-hard functions. 2009.

[70] PIMSLEUR, P. A memory schedule. *Modern Language Journal* (1967), 73–75.

[71] PROCTOR, R. W., LIEN, M.-C., VU, K.-P. L., SCHULTZ, E. E., AND SALVENDY, G. Improving computer security for authentication of users: Influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers 34*, 2 (2002), 163–169.

[72] PROVOS, N., AND MAZIERES, D. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 81–91.

[73] RABKIN, A. Personal knowledge questions for fallback authentication: Security questions in the era of Facebook. In *Proceedings of the 4th Symposium on Usable Privacy and Security* (2008), ACM, pp. 13–23.

[74] ROSS, J., IRANI, L., SILBERMAN, M. S., ZALDIVAR, A., AND TOMLINSON, B. Who Are the Crowdworkers?: Shifting Demographics in Mechanical Turk. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI EA '10, ACM, pp. 2863–2872.

[75] SANCHEZ, D. J., GOBEL, E. W., AND REBER, P. J. Performing the unexplainable: Implicit task performance reveals individually reliable sequence learning without explicit knowledge. *Psychonomic Bulletin & Review 17*, 6 (2010), 790–796.

[76] SCHECHTER, S., BRUSH, A. B., AND EGELMAN, S. It's No Secret. Measuring the Security and Reliability of Authentication via "Secret" Questions. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 375–390.

[77] SHAY, R., KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., UR, B., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Correct horse battery staple: Exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (2012), ACM, p. 7.

[78] STOBERT, E. A. Memorability of Assigned Random Graphical Passwords. Master's thesis, Carleton University, 2011.

[79] STROOP, J. R. Studies of Interference in Serial Verbal Reactions. *Journal of Experimental Psychology 18*, 6 (Dec. 1935), 643–662.

[80] STUBBLEFIELD, A., AND SIMON, D. Inkblot authentication. *Microsoft Research* (2004).

[81] TAO, H., AND ADAMS, C. Pass-Go: A Proposal to Improve the Usability of Graphical Passwords. *IJ Network Security 7*, 2 (2008), 273–292.

[82] THE ETHICAL RESEARCH PROJECT. Post-experiment survey for deception studies. `https://www.ethicalresearch.org/`.

[83] VAN OORSCHOT, P. C., AND THORPE, J. On predictive models and user-drawn graphical passwords. *ACM Transactions on Information and System Security (TISSEC) 10*, 4 (2008), 5.

[84] VAN OORSCHOT, P. C., AND THORPE, J. Exploiting predictability in click-based graphical passwords. *Journal of Computer Security 19*, 4 (2011), 669–702.

[85] VERAS, R., COLLINS, C., AND THORPE, J. On the semantic patterns of passwords and their security impact. In *Network and Distributed System Security Symposium (NDSS'14)* (2014).

[86] VU, K.-P. L., PROCTOR, R. W., BHARGAV-SPANTZEL, A., TAI, B.-L. B., COOK, J., AND EUGENE SCHULTZ, E. Improving password security and memorability to protect personal and organizational information. *International Journal of Human-Computer Studies 65*, 8 (2007), 744–757.

[87] WEINSHALL, D., AND KIRKPATRICK, S. Passwords You'll Never Forget, but Can't Recall. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2004), CHI EA '04, ACM, pp. 1399–1402.

[88] WIEDENBECK, S., WATERS, J., BIRGET, J.-C., BRODSKIY, A., AND MEMON, N. PassPoints: Design and longitudinal evaluation of a graphical password system. *International Journal of Human-Computer Studies 63*, 1 (2005), 102–127.

[89] WIXTED, J. T. The psychology and neuroscience of forgetting. *Annual Psychology Review 55* (2004), 235–269.

[90] WOOD, H. M. *The use of passwords for controlled access to computer resources*, vol. 500. US Department of Commerce, National Bureau of Standards, 1977.

[91] WOZNIAK, P. SuperMemo 2004. *TESL EJ 10*, 4 (2007).

[92] YAN, J. J., BLACKWELL, A. F., ANDERSON, R. J., AND GRANT, A. Password Memorability and Security: Empirical Results. *IEEE Security & privacy 2*, 5 (2004), 25–31.

[93] ZVIRAN, M., AND HAGA, W. User authentication by cognitive passwords: an empirical assessment. In *Proceedings of the 5th Jerusalem Conference on Information Technology* (Oct 1990), pp. 137–144.

[94] ZVIRAN, M., AND HAGA, W. J. Passwords Security: An Exploratory Study. Tech. rep., Naval Postgraduate School, 1990.

[95] ZVIRAN, M., AND HAGA, W. J. Password security: an empirical study. *Journal of Management Information Systems 15* (1999), 161–186.

able	abuse	acid	acorn	acre	actor	add	adobe	adult	aft	age	agile	agony
air	alarm	album	alert	alive	ally	amber	ample	angle	anvil	apply	apron	arbor
area	army	aroma	arrow	arson	ask	aspen	asset	atlas	atom	attic	audit	aunt
aura	auto	aware	awful	axis	baby	back	bad	baker	bare	basis	baton	beam
beer	begin	belly	bench	best	bias	big	birth	bison	bite	blame	blind	bloom
blue	board	body	bogus	bolt	bones	book	born	bound	bowl	box	brain	break
brief	broth	brute	buddy	buff	bugle	build	bulk	burst	butt	buy	buzz	cabin
cadet	call	camp	can	cargo	case	cedar	cello	cent	chair	check	child	chose
chute	cider	cigar	city	civil	class	clear	climb	clock	club	coal	cobra	code
cog	color	comic	copy	cord	cost	court	cover	craft	crew	crime	crown	cruel
cups	curve	cut	cycle	daily	dance	dark	dash	data	death	debt	decoy	delay
depot	desk	diary	diet	dim	ditto	dizzy	dose	doubt	downy	dozen	drawn	dream
drive	drop	drug	dry	due	dust	duty	dwarf	eager	early	easy	eaten	ebb
echo	edge	edit	egg	elbow	elder	elite	elm	empty	end	enemy	entry	envy
equal	era	error	essay	ether	event	exact	exile	extra	eye	fact	faith	false
fancy	far	fatal	fault	favor	feast	feet	fence	ferry	fetch	feud	fever	fiber
field	fifty	film	find	first	fit	flat	flesh	flint	flow	fluid	fly	focus
foe	folk	foot	form	four	foyer	frame	free	front	fruit	full	fume	funny
fused	fuzzy	gala	gang	gas	gauge	gaze	gel	ghost	giant	gift	give	glad
gleam	glory	glut	goat	good	gorge	gourd	grace	great	grid	group	grub	guard
guess	guide	gulf	gym	habit	half	hand	happy	harsh	hasty	haul	haven	hawk
hazy	head	heel	help	hem	here	high	hike	hint	hoax	holy	home	honor
hoop	hot	house	huge	human	hurt	husk	hyper	ice	idea	idle	idol	ill
image	inch	index	inner	input	iris	iron	issue	item	ivory	ivy	jade	jazz
jewel	job	join	joke	jolly	judge	juice	junk	jury	karma	keep	key	kid
king	kiss	knee	knife	known	labor	lady	laid	lamb	lane	lapse	large	last
laugh	lava	law	layer	leaf	left	legal	lemon	lens	level	lies	life	lily
limit	link	lion	lip	liter	loan	lobby	local	lodge	logic	long	loose	loss
loud	love	lowly	luck	lunch	lynx	lyric	madam	magic	main	major	mango	maple
march	mason	may	meat	media	melon	memo	menu	mercy	mess	metal	milk	minor
mixed	model	moist	mole	mom	money	moral	motor	mouth	moved	mud	music	mute
myth	nap	navy	neck	need	neon	new	nine	noble	nod	noise	nomad	north
note	noun	novel	numb	nurse	nylon	oak	oats	ocean	offer	oil	old	one
open	optic	orbit	order	organ	ounce	outer	oval	owner	pale	panic	paper	part
pass	path	pause	pawn	pearl	pedal	peg	penny	peril	petty	phase	phone	piano
piece	pipe	pitch	pivot	place	plea	plot	plug	poet	point	polo	pond	poor
poppy	porch	posse	power	press	price	proof	pub	pulse	pump	pupil	pure	quart
queen	quite	radio	ram	range	rapid	rate	razor	real	rebel	red	reef	relic
rents	reply	resin	rhyme	rib	rich	ridge	right	riot	rise	river	road	robot
rock	roll	room	rope	rough	row	royal	ruby	rule	rumor	run	rural	rush
saga	salt	same	satin	sauce	scale	scene	scope	scrap	sedan	sense	serve	set
seven	sewer	share	she	ship	show	shrub	sick	side	siege	sign	silly	siren
six	skew	skin	skull	sky	slack	sleep	slice	sloth	slump	small	smear	smile
snake	sneer	snout	snug	soap	soda	solid	sonic	soon	sort	soul	space	speak
spine	split	spoke	spur	squad	state	step	stiff	story	straw	study	style	sugar
suit	sum	super	surf	sway	sweet	swift	sword	syrup	taboo	tail	take	talk
taste	tax	teak	tempo	ten	term	text	thank	theft	thing	thorn	three	thumb
tiara	tidal	tiger	tilt	time	title	toast	today	token	tomb	tons	tooth	top
torso	total	touch	town	trade	trend	trial	trout	true	tube	tuft	tug	tulip
tuna	turn	tutor	twist	two	type	ultra	uncle	union	upper	urban	urge	user
usual	value	vapor	vat	vein	verse	veto	video	view	vigor	vinyl	viper	virus
visit	vital	vivid	vogue	voice	voter	vowel	wafer	wagon	wait	waltz	warm	wasp

Table 3: The 676 (26^2) words used by the *words* treatment

Automatically Detecting Vulnerable Websites Before They Turn Malicious

Kyle Soska and Nicolas Christin
Carnegie Mellon University
{ksoska, nicolasc}@cmu.edu

Abstract

Significant recent research advances have made it possible to design systems that can automatically determine with high accuracy the maliciousness of a target website. While highly useful, such systems are reactive by nature. In this paper, we take a complementary approach, and attempt to design, implement, and evaluate a novel classification system which predicts, whether a given, not yet compromised website will become malicious *in the future*. We adapt several techniques from data mining and machine learning which are particularly well-suited for this problem. A key aspect of our system is that the set of features it relies on is automatically extracted from the data it acquires; this allows us to be able to detect new attack trends relatively quickly. We evaluate our implementation on a corpus of 444,519 websites, containing a total of 4,916,203 webpages, and show that we manage to achieve good detection accuracy over a one-year horizon; that is, we generally manage to correctly predict that currently benign websites will become compromised within a year.

1 Introduction

Online criminal activities take many different forms, ranging from advertising counterfeit goods through spam email [21], to hosting "drive-by-downloads" services [29] that surreptitiously install malicious software ("malware") on the victim machine, to distributed denial-of-service attacks [27], to only name a few. Among those, research on analysis and classification of end-host malware – which allows an attacker to take over the victim's computer for a variety of purposes – has been a particularly active field for years (see, e.g., [6, 7, 16] among many others). More recently, a number of studies [8,15,20,22,36] have started looking into "webserver malware," where, instead of targeting arbitrary hosts for compromise, the attacker attempts to inject code on machines running web servers. Webserver malware differs from end-host malware in its design and objectives. Webserver malware indeed frequently exploits outdated or unpatched versions of popular content-management systems (CMS). Its main goal is usually not to completely compromise the machine on which it resides, but instead to get the victimized webserver to participate in search-engine poisoning or redirection campaigns promoting questionable services (counterfeits, unlicensed pharmaceuticals, ...), or to act as a delivery server for malware.

Such infections of webservers are particularly common. For instance, the 2013 Sophos security threat report [33, p.7] states that in 2012, 80% of websites hosting malicious contents were compromised webservers that belonged to unsuspecting third-parties. Various measurement efforts [20, 25, 36] demonstrate that people engaging in the illicit trade of counterfeit goods are increasingly relying on compromised webservers to bring traffic to their stores, to the point of supplanting spam as a means of advertising [20].

Most of the work to date on identifying webserver malware, both in academia (e.g., [8, 15]) and industry (e.g., [3, 5, 14, 24]) is primarily based on detecting the presence of an *active infection* on a website. In turn, this helps determine which campaign the infected website is a part of, as well as populating blacklists of known compromised sites. While a highly useful line of work, it is by design reactive: only websites that have already been compromised can be identified.

Our core contribution in this paper is to propose, implement, and evaluate a general methodology to identify webservers that are at a high risk of becoming malicious *before* they actually become malicious. In other words, we present techniques that allow to proactively identify likely targets for attackers as well as sites that may be hosted by malicious users. This is particularly useful for search engines, that need to be able to assess whether or not they are linking to potentially risky contents; for blacklist operators, who can obtain, ahead of time, a list of sites to keep an eye on, and potentially warn these

sites' operators of the risks they face ahead of the actual compromise; and of course for site operators themselves, which can use tools based on the techniques we describe here as part of a good security hygiene, along with practices such as penetration testing.

Traditional penetration testing techniques often rely on ad-hoc procedures rather than scientific assessment [26] and are greatly dependent on the expertise of the tester herself. Different from penetration testing, our approach relies on an online classification algorithm ("classifier") that can 1) automatically detect whether a server is likely to become malicious (that is, it is probably vulnerable, and the vulnerability is actively exploited in the wild; or the site is hosted with malicious intent), and that can 2) quickly adapt to emerging threats. At a high level, the classifier determines if a given website shares a set of features (e.g., utilization of a given CMS, specifics of the webpages' structures, presence of certain keywords in pages, ...) with websites known to have been malicious. A key aspect of our approach is that the feature list used to make this determination is automatically extracted from a training set of malicious and benign webpages, and is updated over time, as threats evolve.

We build this classifier, and train it on 444,519 archives sites containing a total of 4,916,203 webpages. We are able to correctly predict that sites will eventually become compromised within 1 year while achieving a true positive rate of 66% and a false positive rate of 17%. This level of performance is very encouraging given the large imbalance in the data available (few examples of compromised sites as opposed to benign sites) and the fact that we are essentially trying to predict the future. We are also able to discover a number of content features that were rather unexpected, but that, in hindsight, make perfect sense.

The remainder of this paper proceeds as follows. We review background and related work in Section 2. We detail how we build our classifier in Section 3, describe our evaluation and measurement methodology in Section 4, and present our empirical results in Section 5. We discuss limitations of our approach in Section 6 before concluding in Section 7.

2 Background and related work

Webserver malware has garnered quite a bit of attention in recent years. As part of large scale study on spam, Levchenko et al. [21] briefly allude to search-engine optimization performed my miscreants to drive traffic to their websites. Several papers [17, 19, 20, 22] describe measurement-based studies of the "search-redirection" attacks, in which compromised websites are first being used to link to each other and associate themselves with searches for pharmaceutical and illicit products; this allows the attacker to have a set of high-ranked links displayed by the search engine in response to such queries. The second part of the compromise is to have a piece of malware on the site that checks the provenance of the traffic coming to the compromise site. For instance, if traffic is determined to come from a Google search for drugs, it is immediately redirected—possibly through several intermediaries—to an illicit online pharmacy. These studies are primarily empirical characterizations of the phenomenon, but do not go in great details about how to curb the problem from the standpoint of the compromised hosts.

In the same spirit of providing comprehensive measurements of web-based abuse, McCoy et al. [25] looks at revenues and expenses at online pharmacies, including an assessment of the commissions paid to "network affiliates" that bring customers to the websites. Wang et al. [36] provides a longitudinal study of a search-engine optimization botnet.

Another, recent group of papers looks at how to detect websites that have been compromised. Among these papers, Invernizzi et al. [15] focuses on automatically finding recently compromised websites; Borgolte et al. [8] look more specifically at previously unknown web-based infection campaigns (e.g., previously unknown injections of obfuscated JavaScript-code). Different from these papers, we use machine-learning tools to attempt to detect websites that have not been compromised yet, but that are likely to become malicious in the future, over a reasonably long horizon (approximately one year).

The research most closely related to this paper is the recent work by Vasek and Moore [35]. Vasek and Moore manually identified the CMS a website is using, and studied the correlation between that CMS the website security. They determined that in general, sites using a CMS are more likely to behave maliciously, and that some CMS types and versions are more targeted and compromised than others. Their research supports the basic intuition that the content of a website is a coherent basis for making predictions about its security outcome.

This paper builds on existing techniques from machine learning and data mining to solve a security issue. Directly related to the work we present in this paper is the data extraction algorithm of Yi et al. [38], which we adapt to our own needs. We also rely on an ensemble of decision-tree classifiers for our algorithm, adapting the techniques described by Gao et al. [13].

3 Classifying websites

Our goal is to build a classifier which can predict with high certainty if a given website will become malicious in the future. To that effect, we start by discussing the properties our classifier must satisfy. We then elaborate on the learning process our classifier uses to differentiate between benign and malicious websites. Last, we de-

scribe an automatic process for selecting a set features that will be used for classification.

3.1 Desired properties

At a high level, our classifier must be efficient, interpretable, robust to imbalanced data, robust to missing features when data is not available, and adaptive to an environment that can drastically change over time. We detail each point in turn below.

Efficiency: Since our classifier uses webpages as an input, the volume of the data available to train (and test) the classifier is essentially the entire World Wide Web. As a result, it is important the the classifier scale favorably with large, possibly infinite datasets. The classifier should thus use an online learning algorithm for learning from a streaming data source.

Interpretability: When the classifier predicts whether a website will become malicious (i.e., it is vulnerable, and likely to be exploited; or likely to host malicious content), it is useful to understand why and how the classifier arrived at the prediction. Interpretable classification is essential to meaningfully inform website operators of the security issues they may be facing. Interpretability is also useful to detect evolution in the factors that put a website at risk of being compromised. The strong requirement for interpretability unfortunately rules out a large number of possible classifiers which, despite achieving excellent classification accuracy, generally lack interpretability.

Robustness to imbalanced data: In many applications of learning, the datasets that are available are assumed to be balanced, that is, there is an equal number of examples for each class. In our context, this assumption is typically violated as examples of malicious behavior tend to be relatively rare compared to innocuous examples. We will elaborate in Section 5 on the relative sizes of both datasets, but assume, for now, that 1% of all existing websites are likely to become malicious, i.e., they are vulnerable, and exploits for these vulnerabilities exist and are actively used; or they are hosted by actors with malicious intent. A trivial classifier consistently predicting that all websites are safe would be right 99% of the time! Yet, it would be hard to argue that such a classifier is useful at all. In other words, our datasets are imbalanced, which has been shown to be problematic for learning—the more imbalanced the datasets, the more learning is impacted [30].

At a fundamental level, simply maximizing accuracy is not an appropriate performance metric here. Instead, we will need to take into account both false positives (a benign website is incorrectly classified as vulnerable) and false negatives (a vulnerable website is incorrectly classified as benign) to evaluate the performance of our classifier. For instance, the trivial classifier discussed above, which categorizes all input as benign, would yield 0% false positives, which is excellent, but 100% of false negatives among the population of vulnerable websites, which is obviously inadequate. Hence, metrics such as receiver-operating characteristics (ROC) curves which account for both false positive and false negatives are much more appropriate in the context of our study for evaluating the classifier we design.

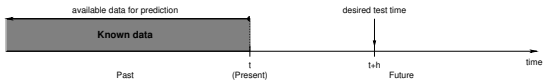
Robustness to errors: Due to its heterogeneity (many different HTML standards co-exist, and HTML engines are usually fairly robust to standard violations) and its sheer size (billions of web pages), the Web is a notoriously inconsistent dataset. That is, for any reasonable set of features we can come up with, it will be frequently the case that some of the features may either be inconclusive or undetermined. As a simple example, imagine considering website popularity metrics given by the Alexa Web Information Service (AWIS, [1]) as part of our feature set. AWIS unfortunately provides little or no information for very unpopular websites. Given that webpage popularity distribution is "heavy-tailed [9]," these features would be missing for a significant portion of the entire population. Our classifier should therefore be robust to errors as well as missing features.

Another reason for the classifier to be robust to errors is that the datasets used in predicting whether a website will become compromised are fundamentally noisy. Blacklists of malicious websites are unfortunately incomplete. Thus, malicious sites may be mislabeled as benign, and the classifier's performance should not degrade too severely in the presence of mislabeled examples.

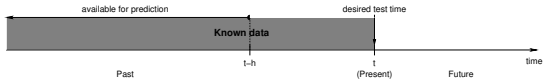
Adaptive: Both the content on the World Wide Web, and the threats attackers pose vary drastically over time. As new exploits are discovered, or old vulnerabilities are being patched, the sites being attacked change over time. The classifier should thus be able to learn the evolution of these threats. In machine learning parlance, we need a classifier that is adaptive to "concept drift" [37].

All of these desired properties led us to consider an ensemble of decision-tree classifiers. The method of using an ensemble of classifiers is taken from prior work [13]. The system works by buffering examples from an input data stream. After a threshold number of examples has been reached, the system trains a set of classifiers by resampling all past examples of the minority class as well as recent examples of the majority class. While the type of classifier used in the ensemble may vary, we chose to use C4.5 decision trees [31].

The system is efficient as it does not require the storage or training on majority class examples from the far past. The system is also interpretable and robust to errors as the type of classifier being used is a decision-tree in



(a) Using the present to predict the future



(b) Using the past to predict the present

Figure 1: **Prediction timeline.** Attempting to predict the future makes it impossible to immediately evaluate whether the prediction was correct (a). A possible alternative (b) is to use past data to simulate a prediction done in the past (at $t-h$) that can then be tested at the present time t.

an ensemble [13]. Periodically retraining our classifiers makes them robust to concept drift as long as retraining occurs sufficiently often. Finally, the system handles class imbalance by resampling the input stream, namely, it resamples from the set of all minority class training examples from the past as well as recent majority class examples.

3.2 Learning process

The type of classification we aim to perform presents unique challenges in the learning process.

Lack of knowledge of the future: Assume that at a given time t, our classifier predicts that a given website w is likely to become compromised in the future. Because the website has not been compromised yet—and may not be compromised for a while—we cannot immediately know whether the prediction is correct. Instead, we have to wait until we have reached a time $(t+h)$ to effectively be able to verify whether the site has become compromised between t and $(t+h)$, or if the classifier was in error. This is particularly problematic, since just training the classifier—let alone using it—would require to wait at least until $(t+h)$. This is the situation illustrated in Figure 1(a).

A second, related issue, is that of defining a meaningful "time horizon" h. If h is too long, it will be impossible to even verify that the classifier was right. In an extreme case, when $h \to \infty$, the performance of the classifier cannot be evaluated.[1] Selecting a time horizon too short (e.g., $h = 0$) would likewise reduce to the problem of determining whether a website is already compromised or not—a very different objective for which a rich literature already exists, as discussed earlier.

[1] Given the complexity of modern computer software, it is likely that exploitable bugs exist in most, if not all webservers, even though they might have not been found yet. As a result, a trivial classifier predicting that all websites will be compromised over an infinite horizon ($h \to \infty$) may not even be a bad choice.

We attempt to solve these issues as follows. First, deciding what is a meaningful value for the horizon h appears, in the end, to be a design choice. Unless otherwise noted, we will assume that h is set to one year. This choice does not affect our classifier design, but impacts the data we use for training.

Second, while we cannot predict the future at time t, we can use the past for training. More precisely, for training purposes we can solve our issue if we could extract a set of features, and perform classification on an archived version of the website w as it appeared at time $(t-h)$ and check whether, by time t, w has become malicious. This is what we depict in Figure 1(b). Fortunately, this is doable: At the time of this writing, the Internet Archive's Wayback Machine [34] keeps an archive of more than 391 billion webpages saved over time, which allows us to obtain "past versions" of a large number of websites.

Obtaining examples of malicious and benign websites: To train our classifier, we must have ground truth on a set of websites—some known to be malicious, and some known to be benign. Confirmed malicious websites can be obtained from blacklists (e.g., [28]). In addition, accessing historical records of these blacklists allows us to determine (roughly) at what time a website became malicious. Indeed, the first time at which a compromised website appeared in a blacklist gives an upper bound on the time at which the site became malicious. We can then grab older archived versions of the site from the Wayback Machine to obtain an example of a site that was originally not malicious and then became malicious.

We obtain benign websites by randomly sampling DNS zone files, and checking that the sampled sites are not (and have never been) in any blacklist. We then also cull archives of these benign sites from the Wayback machine, so that we can compare *at the same time in the past* sites that have become malicious to sites that have remained benign.

We emphasize that, to evaluate the performance of the classifier at a particular time t, training examples from the past (e.g., $t-h$) may be used; and these examples can then be used to test on the future. However, the converse is not true: even if that data is available, we cannot train on the present t and test on the past $(t-h)$ as we would be using future information that was unknown at the time of the test. Figure 1(b) illustrates that data available to build predictions is a strict subset of the known data.

Dealing with imbalanced datasets: As far as the learning process is concerned, one can employ class re-balancing techniques. At a high level, class re-balancing has been studied as a means to improve classifier performance by training on a distribution other than the naturally sampled distribution. Since we sample only a random subset of sites which were not compromised, we

already perform some resampling in the form of a one-sided selection.

3.3 Dynamic extraction of the feature list

Any classifier needs to use a list of features on which to base its decisions. Many features can be used to characterize a website, ranging from look and feel, to traffic, to textual contents. Here we discuss in more details these potential features. We then turn to a description of the dynamic process we use to update these features.

3.3.1 Candidate feature families

As potential candidates for our feature list, we start by considering the following families of features.

Traffic statistics. Website statistics on its traffic, popularity, and so forth might be useful in indicating a specific website became compromised. For instance, if a certain website suddenly sees a change in popularity, it could mean that it became used as part of a redirection campaign. Such statistics may be readily available from services such as the aforementioned Alexa Web Information Service, if the website popularity is not negligible.

Filesystem structure. The directory hierarchy of the site, the presence of certain files may all be interesting candidate features reflecting the type of software running on the webserver. For instance the presence of a `wp-admin` directory might be indicative of a specific content management system (WordPress in that case), which in turn might be exploitable if other features indicate an older, unpatched version is running.

Webpage structure and contents. Webpages on the website may be a strong indicator of a given type of content-based management system or webserver software. To that effect, we need to distill useful page structure and content from a given webpage. The user-generated content within webpages is generally not useful for classification, and so it is desirable to filter it out and only keep the "template" the website uses. Extracting such a template goes beyond extraction of the Document Object Model (DOM) trees, which do not provide an easy way to differentiate between user-generated contents and template. We discuss in the next section how extracting this kind of information can be accomplished in practice.

Page content can then distilled into features using several techniques. We chose to use binary features that detect the presence of particular HTML tags in a site. For instance, "is the keyword *joe's guestbook/v1.2.3* present?" is such a binary feature. Of course, using such a binary encoding will result in a rather large feature set as it is less expressive than other encoding choices. However the resulting features are extremely interpretable and, as we will see later, are relatively straightforward to extract automatically.

Perhaps more interestingly, we observed that features on filesystem structure can actually be captured by looking at the contents of the webpages. Indeed, when we collect information about internal links (e.g., *<a href="../top.html">*) we are actually gathering information about the filesystem as well. In other words, features characterizing the webpage structure provide enough information for our purposes.

3.3.2 Dynamic updates

We consider traffic statistics as "static" features that we always try to include in the classification process, at least when they are available. On the other hand, all of the content-based features are dynamically extracted. We use a statistical heuristic to sort features which would have been useful for classifying recent training examples and apply the top performing features to subsequent examples.

4 Implementation

We next turn to a discussion of how we implemented our classifier in practice. We first introduce the data sources we used for benign and soon-to-be malicious websites. We then turn to explaining how we conducted the parsing and filtering of websites. Last we give details of how we implemented dynamic feature extraction.

4.1 Data sources

We need two different sources of data to train our classifier: a ground truth for soon-to-be malicious websites, and a set of benign websites.

Malicious websites. We used two sets of blacklists as ground truth for malicious websites. First, we obtained historical data from PhishTank [28]. This data contains 11,724,276 unique links from 91,155 unique sites, collected between February 23, 2013 and December 31, 2013. The Wayback machine contained usable archives for 34,922 (38.3%) of these sites within the required range of dates.

We then complemented this data with a list of websites known to have been infected by "search-redirection attacks," originally described in 2011 [20, 22]. In this attack, miscreants inject code on webservers to have them participate in link farming and advertise illicit products—primarily prescription drugs. From a related measurement project [19], we obtained a list, collected between October 20, 2011 and September 16, 2013, of 738,479 unique links, all exhibiting redirecting behavior, from 16,173 unique sites. Amazingly, the Wayback machine contained archives in the acceptable range for 14,425 (89%) of these sites.

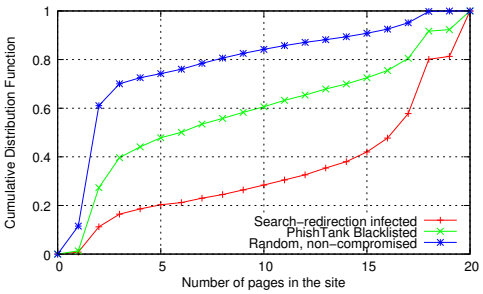


Figure 2: **Cumulative distribution function of the number of pages scraped.** Benign websites very frequently contain only a handful of pages.

We use these two blacklists in particular because we determined, through manual inspection, that a large percentage of sites in these lists have either been compromised by an external attacker or are maliciously hosted. On the other hand, various other blacklists often label sites that are heavily spammed or contain adult content as malicious, which we do not think is appropriate.

Benign websites. We randomly sampled the entire `.com` zone file from January 14th, 2014. For each domain, we enumerated the available archives in the Wayback machine. If at least an archive was found, we selected one of the available archives in the range of February, 20, 2010 to September 31, 2013. This yielded 337,191 website archives. We then removed all archives that corresponded to sites known as malicious. We removed 27 of them that were among the set of sites known to have been infected by search-redirection attacks, and another 72 that matched PhishTank entries. We also discarded an additional 421 sites found in the DNS-BH [2], Google SafeBrowsing [14], and hpHosts [23] blacklists, eventually using 336,671 websites in our benign corpus.

Structural properties. Figure 2 shows some interesting characteristics of the size of the websites we consider. Specifically, the cumulative distribution function of the number of pages each website archive contains differs considerably between the datasets. For many benign sites, that were randomly sampled from zone files, only a few pages were archived. This is because many domains host only a parking page or redirect (without being malicious) to another site immediately. Other sites are very small and host only a few different pages.

On the other hand, malicious sites from both of our blacklists contain more pages per site, since in many cases they are reasonably large websites that had some form of visibility (e.g., in Google rankings), before becoming compromised and malicious. In some other cases, some of the blacklisted sites are sites maliciously registered, that do host numerous phishing pages.

4.2 Parsing and filtering websites

We scraped web pages from the Wayback Machine using the Scrapy framework [4], and a collection of custom Python scripts.

Selecting which archive to use. The scripts took in a URL and a range of dates as inputs, and then navigated The WayBack Machine to determine all the archives that existed for that URL within the specified range.

Sites first appear in a blacklist at a particular time t. If a site appears in multiple blacklists or in the same blacklist multiple times, we use the earliest known infection date. We then search for snapshots archived by the Wayback machine between $t-12$ months and to $t-3$ months prior to the site being blacklisted. The choice of this range is to satisfy two concerns about the usefulness of the archive data. Because compromised sites are not generally instantaneously detected, if the date of the archive is chosen too close to the first time the site appeared in a blacklist, is is possible that the archived version was already compromised. On the other hand, if the archived version was chosen too far from the time at which the site was compromised, the site may have changed dramatically. For instance, the content management system powering the site may have been updated or replaced entirely.

If multiple archives exist in the range $t-3$ months–$t-12$ months, then we select an archive as close to $t-12$ months as possible; this matches our choice for $h=1$ year described earlier. We also download and scrape the most recent available archive, and compare it with the the one-year old archive to ensure that they are using the same content management system. In the event that the structure of the page has changed dramatically (defined as more than 10% changes) we randomly select a more recent archive (i.e., between zero and one year old), and repeat the process.

Scraping process. We scrape each archived site, using a breadth-first search. We terminate the process at either a depth of two links, or 20 pages have been saved, and purposefully only download text (HTML source, and any script or cascading style sheets (CSS) embedded in the page, but no external scripts or images). Using a breadth-first search allows us to rapidly sample a large variety of web pages. It is indeed common for websites to contain multiple kinds of webpages, for example forums and posts, blogs and guestbooks, login and contact pages. A breadth-first search provides an idea of the amount of page diversity in a site without requiring us to scrape the entire site. Limiting ourselves to 20 pages allows us to quickly collect information on a large number of websites, and in fact allows us to capture the vast majority of websites in their entirety, according to Figure 2,

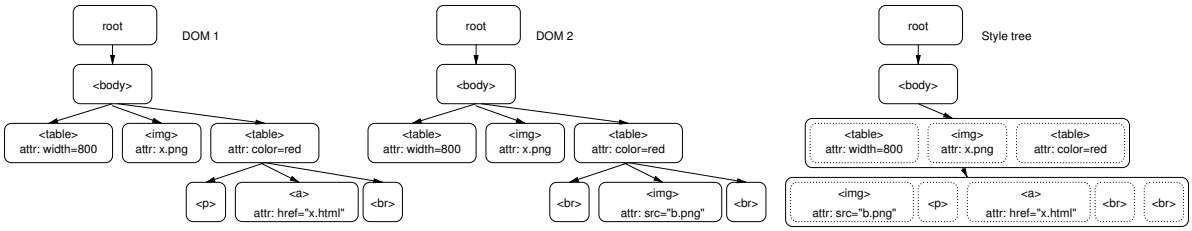


Figure 3: **DOM and style trees**. The figure, adapted from Yi et al. [38], shows two DOM trees corresponding to two separate pages, and the resulting style tree.

while—as we will see later—providing enough information to our classifier to be able to make good decisions.

Filtering. Once a batch of webpages has been saved for a given website, we filter each of them to remove user-generated content. We define user-generated content as all data in webpage, visible and invisible, which is not part of the underline template or content-management system. This includes for instance blog posts, forum posts, guestbook entries, and comments. Our assumption is that user-generated content is orthogonal to the security risks that a site a priori faces and is therefore simply noise to the classifier. User-generated content can, on the other hand, indicate that a site has *already* been compromised, for instance if blog posts are riddled with spam links and keywords. But, since our objective is to detect vulnerable (as opposed to already compromised) sites, user-generated content is not useful to our classification.

The process of extracting information from webpages is a well-studied problem in data mining [10, 11, 32, 38, 39]. Generally the problem is framed as attempting to isolate user-generated content which otherwise would be diluted by page template content. We are attempting to do the exact opposite thing: discarding user-generated content while extracting templates. To that effect, we "turn on its head" the content-extraction algorithm proposed by Yi et al. [38] to have it only preserve templates and discard contents.

Yi et al. describes an algorithm where each webpage in a website is broken down into a Document Object Model (DOM) tree and joined into a single larger structure referred to as a style tree. We illustrate this construction in Figure 3. In the figure, two different pages in a given website produce two different DOM trees (DOM 1 and DOM 2 in the figure). DOM trees are essentially capturing the tags and attributes present in the page, as well as their relationship; for instance, *¡table¿* elements are under *¡body¿*.

The style tree incorporates not only a summary of the individual pieces of content within the pages, but also their structural relationships with each other. Each node in a style tree represents an HTML tag from one or possibly many pages within the site and has a derived property called *composite importance* (CompImp), which is an information-based measure of how important the node is. Without getting into mathematical details—which can be found in Yi et al. [38]—nodes which have a CompImp value close to 1 are either unique, or have children which are unique. On the other hand, nodes which have a CompImp value closer to 0 typically appear often in the site.

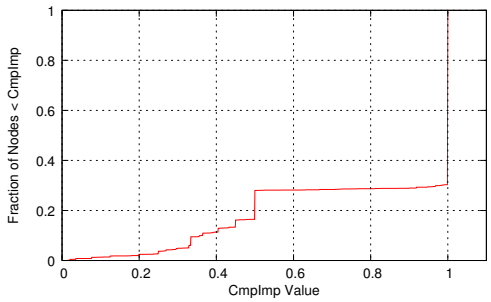
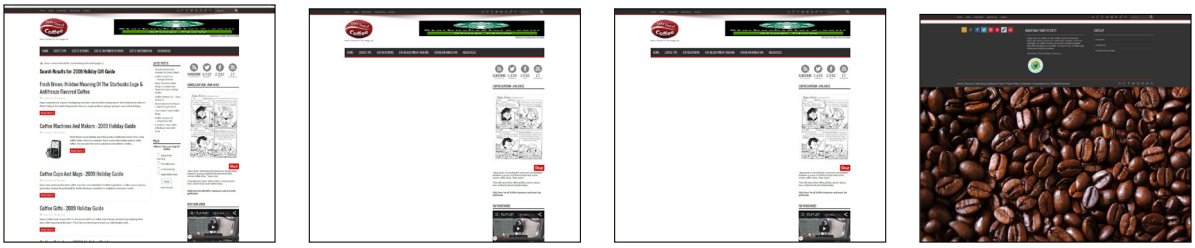


Figure 4: **C.d.f. of CompImp over the style tree generated for `dailyshotofcoffee.com`**. We use 1,000 pages to generate the style tree.

While Yi et al. try to filter out nodes with CompImp below a given threshold to extract user content, we are interested in the exact opposite objective; so instead we filter out nodes whose CompImp is *above* a threshold.

We provide an illustration with the example of `dailyshotofcoffee.com`. We built, for the purpose of this example, a style tree using 1,000 randomly sampled pages from this website. We plot the CompImp of nodes in the resulting style tree in Figure 4. A large portion of the nodes in the style tree have a CompImp value of exactly 1 since their content is completely unique within the site. The jumps in the graph show that some portions of the site may use different templates or different variations of the same template. For example, particular navigation bars are present when viewing some pages but not when viewing others.

We illustrate the effect of selecting different values as a threshold for filtering in Figure 5. We consider a ran-



(a) Original (b) CompImp > 0.99 (c) CompImp > 0.6 (d) CompImp > 0.1

Figure 5: **Impact of various thresholds on filtering.** The figure shows how different CmpInt thresholds affect the filtering of the webpage shown in (a). Thresholds of 0.99 and 0.6 produce the same output, whereas a threshold of 0.1 discards too many elements.

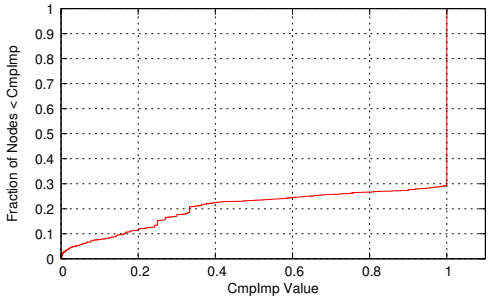


Figure 6: **C.d.f. of CompImp over all style trees generated for 10,000 random sites**. We use 1,000 pages per website to generate the style tree.

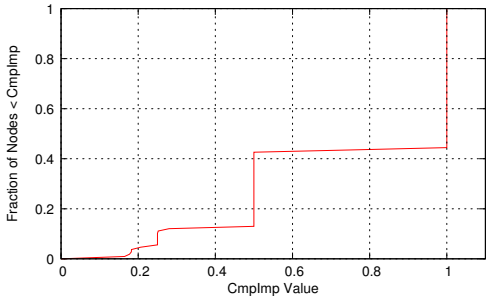


Figure 7: **C.d.f. of CompImp over the style tree generated for `dailyshotofcoffee.com` using only 5 pages.** The plot represents the fraction of nodes less than a threshold in the style tree generated from only five random pages from `dailyshotofcoffee.com`.

dom page of `dailyshotofcoffee.com` showed in Figure 5(a). In Figures 5(b), (c), and (d), we show the result of filtering with a threshold value of 0.99, 0.6 and 0.1 respectively. There is no difference between using 0.99 and 0.6 as a threshold since there are very few nodes in the style tree that had a CompImp between 0.6 and 0.99, as shown in Figure 4. There is a notable difference when using 0.1 as a threshold since portions of the page template are present on some but not all pages of the site.

In general, style trees generated for other sites seem to follow a similar distribution, as shown in Figure 6 where we plot the aggregated CompImp c.d.f. over all style trees generated for 10,000 sites. The aggregation does have a slight curve around the CompImp value 1 which indicates that a few sites do have style trees with nodes in this space. Such sites typically use a fixed template with the exception of a few pages such as 404 error pages, login, and registration pages.

A concern with applying style trees for filtering in this setting occurs in instances where there are only a few examples of pages from a particular site. Trivially, if there is only a single page from the site, then the style tree is just the page itself, and if only a few examples are found then the estimates of nodes in the style tree will be highly dependent on where in the site those pages were sampled from. In Figure 7, we plot the cumulative distribution function for CompImp over the style tree generated for `dailyshotofcoffee.com`, but this time, only using five random pages from the site. Compared to Figure 4, we see that the particular cutoffs are slightly different from when we used 1,000 pages; but the general shape still remains the same. Manual inspection over many sites has indicates that this approach still works well with as few as five pages. This serves as further justification to our design decision of only scraping 20 pages at most from each website.

With all of these experiments in mind, we selected a thresholding value of 0.99 for our system, and eliminated from our filtering process sites where we could only scrape less than five pages.

4.3 Feature extraction

We derived the set of features used in classification from two main sources, the Alexa Web Information Service

Feature	Discretization fn.	Values
AWIS Site Rank	$\lceil \log(\text{SiteRank}) \rceil$	$[0 \ldots 8]$
Links to the site	$\lceil \log(\text{LinksIn}) \rceil$	$[0 \ldots 7]$
Load percentile	$\lfloor \text{LoadPercentile}/10 \rfloor$	$[0 \ldots 10]$
Adult site?	(Boolean)	$\{0,1\}$
Reach per million	$\lceil \log(\text{reachPerMillion}) \rceil$	$[0 \ldots 5]$

Table 1: **AWIS features used in our classifier and their discretization.** Those features are static—i.e., they are used whenever available.

(AWIS) and the content of the saved web pages. Features generated from the pages content are dynamically generated during the learning process according to a chosen statistic. For the classification process we use an ensemble of decision trees so that the input features should be discrete. In order to obtain useful discrete features (i.e., discrete features that do not take on too many values relative to the number of examples), we apply a mapping process to convert continuous quantities (load percentile) and large discrete quantities (global page rank) to a useful set of discrete quantities. The mappings used are shown in Table 1.

4.3.1 AWIS features

For every site that was scraped, we downloaded an entry from AWIS on Feb 2, 2014. While the date of the scrape does not match the date of the web ranking information, it can still provide tremendous value in helping to establish the approximate popularity of a site. Indeed, we observed that in the overwhelming majority of cases, the ranking of sites and the other information provided does not change significantly over time; and after discretization does not change at all. This mismatch is not a fundamental consequence of the experiment design but rather a product retrospectively obtaining negative training examples (sites which did not become compromised) which can be done in real time.

Intuitively, AWIS information may be useful because attackers may target their resources toward popular hosts running on powerful hardware. Adversaries which host malicious sites may have incentives to make their own malicious sites popular. Additionally, search engines are a powerful tool used by attackers to find vulnerable targets (through, e.g., "Google dorks [18]") which causes a bias toward popular sites.

We summarize the AWIS features used in Table 1. An AWIS entry contains estimates of a site's global and regional popularity rankings. The entry also contains estimates of the reach of a site (the fraction of all users that are exposed to the site) and the number of other sites which link in. Additionally, the average time that it takes users to load the page and some behavioral measurements such as page views per user are provided.

The second column of Table 1 shows how AWIS information is discretized to be used as a feature in a decision-tree classifier. Discretization groups a continuous feature such as load percentile or a large discrete feature such as global page rank into a few discrete values which make them more suitable for learning. If a feature is continuous or if too many discrete values are used, then the training examples will appear sparse in the feature space and the classifier will see new examples as being unique instead of identifying them as similar to previous examples when making predictions.

For many features such as AWIS Site Rank, a logarithm is used to compress a large domain of ranks down to a small range of outputs. This is reasonable since for a highly ranked site, varying by a particular number of rankings is significant relative to much lower ranked site. This is because the popularity of sites on the Internet follows a long tailed distribution [9].

Dealing with missing features. Some features are not available for all sites, for example information about the number of users reached by the site per million users was not present for many sites. In these cases, there are two options. We could reserve a default value for missing information; or we could simply not provide a value and let the classifier deal with handling missing attributes.

When a decision-tree classifier encounters a case of a missing attribute, it will typically assign it either the most common value for that attribute, the most common value given the target class of the example (when training), or randomly assign it a value based on the estimated distribution of the attribute. In our particular case, we observed that when a feature was missing, the site also tended to be extremely unpopular. We asserted that in these cases, a feature such as reach per million would probably also be small and assigned it a default value. For other types of missing attributes such as page load time, we did not assign the feature a default value since there is likely no correlation between the true value of the attribute and its failure to appear in the AWIS entry.

4.3.2 Content-based features

The content of pages in the observed sites provides extremely useful information for determining if the site will become malicious. Unlike many settings where learning is applied, the distribution of sites on the web and the attacks that they face vary over time.

Many Internet web hosts are attacked via some exploit to a vulnerability in a content-management system (CMS), that their hosted site is using. It is quite common for adversaries to enumerate vulnerable hosts by looking for CMSs that they can exploit. Different CMSs and even different configurations of the same CMS leak information about their presence through content such as tags associated with their template, meta tags, and com-

ments. The set of CMSs being used varies over time: new CMSs are released and older ones fall out of favor, as a result, the page content that signals their presence is also time varying.

To determine content-based features, each of the pages that survived the acquisition and filtering process described earlier was parsed into a set of HTML tags. Each HTML tag was represented as the tuple (type, attributes, content). The tags from all the pages in a site were then aggregated into a list *without repetition*. This means that duplicate tags, i.e., tags matching precisely the same type, attributes and content of another tag were dropped. This approach differs from that taken in typical document classification techniques where the document frequency of terms is useful, since for example a document that uses the word "investment" a dozen times may be more likely to be related to finance than a document that uses it once. However, a website that has many instances of the same tag may simply indicate that the site has many pages. We could balance the number of occurrences of a tag by weighting the number of pages used; but then, relatively homogeneous sites where the same tag appears on every page would give that tag a high score while less homogeneous sites would assign a low score. As a result, we chose to only use the existence of a tag within a site but not the number of times the tag appeared.

During the training phase, we then augmented the lists of tags from each site with the sites' classification; and added to a dictionary which contains a list of all tags from all sites, and a count of the number of positive and negative sites a particular tag has appeared in. This dictionary grew extremely quickly; to avoid unwieldy increase in its size, we developed the following heuristic. After adding information from every 5,000 sites to the dictionary, we purged from the dictionary all tags that had appeared only once. This heuristic removed approximately 85% of the content from the dictionary every time it was run.

Statistic-based extraction. The problem of feature extraction reduces to selecting the particular tags in the dictionary that will yield the best classification performance on future examples. At the time of feature selection, the impact of including or excluding a particular feature is unknown. As a result, we cannot determine an optimal set of features at that time. So, instead we use the following technique. We fix a number N of features we want to use. We then select a statistic $\hat{s}$, and, for each tag t in the dictionary, we compute its statistic $\hat{s}(t)$. We then simply take the top-N ranked entries in the dictionary according to the statistic $\hat{s}$.

Many statistics can be used in practice [12]. In our implementation, we use $N = 200$, and $\hat{s}$ to be ACC2. ACC2 is the balanced accuracy for tag x, defined as:

$$\hat{s}(x) = \left| \frac{|\{x : x \in w, w \in \mathscr{M}\}|}{|\mathscr{M}|} - \frac{|\{x : x \in w, w \in \mathscr{B}\}|}{|\mathscr{B}|} \right|,$$

where $\mathscr{B}$ and $\mathscr{M}$ are the set of benign, and malicious websites, respectively; the notation $x \in w$ means (by a slight abuse of notation) that the tag x is present in the tag dictionary associated with website w. In essence, the statistic computes the absolute value of the difference between the tag frequency in malicious pages and the tag frequency in benign pages.

A key observation is that these top features can be periodically recomputed in order to reflect changes in the statistic value that occurred as a result of recent examples. In our implementation, we recomputed the top features every time that the decision tree classifiers in the ensemble are trained.

As the distribution of software running on the web changes and as the attacks against websites evolve, the tags that are useful for classification will also change. A problem arises when the dictionary of tags from previous examples is large. For a new tag to be considered a top tag, it needs to be observed a large number of times since $|\mathscr{M}|$ and $|\mathscr{B}|$ are very large. This can mean that there is a significant delay between when a tag becomes useful for classification and when it will be selected as a top feature, or for example in the case of tags associated with unpopular CMSs, which will never be used.

A way of dealing with this problem is to use windowing, where the dictionary of tags only contains entries from the last K sites. By selecting a sufficiently small window, the statistic for a tag that is trending can rapidly rise into the top N tags and be selected as a feature. The trade-off when selecting window size is that small window sizes will be less robust to noise but faster to capture new relevant features while larger windows will be more robust to noise and slower to identify new features.

An additional strategy when calculating the statistic value for features is to weight occurrences of the feature differently depending on when they are observed. With windowing, all observations in the window are weighted equally with a coefficient of 1, and all observations outside of the window are discarded by applying a coefficient of 0. Various functions such as linear and exponential may be used to generate coefficients that scale observations and grant additional emphasis on recent observations of a feature.

5 Experimental results

We evaluate here both our dynamic feature extraction algorithm, and the overall performance of our classifier, by providing ROC curves.

Feature	Stat.
meta{'content': 'WordPress 3.2.1', 'name': 'generator'}	0.0569
ul{'class': ['xoxo', 'blogroll']}	0.0446
You can start editing here.	0.0421
meta{'content': 'WordPress 3.3.1', 'name': 'generator'}	0.0268
/all in one seo pack	0.0252
span{'class': ['breadcrumbs', 'pathway']}	0.0226
If comments are open, but there are no comments.	0.0222
div{'id': 'content_disclaimer'}	0.0039

Table 2: **Selection of the top features after processing the first 90,000 examples.** These features are a chosen subset of the top 100 features determined by the system after 90,000 examples had been observed and using windowing with a window size of 15,000 examples and linear attenuation.

5.1 Dynamic Feature Extraction

We analyzed dynamic features by logging the values of the statistic AAC2 after adding every example to the system. We selected a few particular features from a very large set of candidates to serve as examples and to guide intuition regarding dynamic feature extraction. The process of feature extraction could be performed independently of classification and was run multiple times under different conditions to explore the effect of different parameters such as the use of windowing and attenuation.

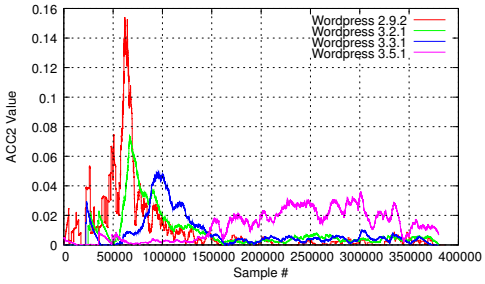


Figure 8: **Statistic value for various tags corresponding to different version of the Wordpress content management system.**

Dynamic feature extraction is essential, as it allows the system to automatically identify features useful in predicting if a domain will become malicious; this automation is imperative in a concept-drifting domain. For example, Figure 8 shows the computed statistic value for various features that correspond directly to different versions of the Wordpress CMS. Over time, the usefulness of different features changes. In general, as new versions of a CMS are released, or new exploits are found for existing ones, or completely new CMSs are developed, the set of the features most useful for learning will be constantly evolving.

Table 2 shows a selection of the 200 tags with highest statistic value after 90,000 examples had been passed to the system using a window size of 15,000 examples and a linear weighting scheme. A meaningful feature, i.e., with a large statistic value, is either a feature whose presence is relatively frequent among examples of malicious sites, or whose presence is frequent among benign sites. Of the 15,000 sites in the window used for generating the table, there were 2,692 malicious sites, and 12,308 benign ones. The feature *ul{'class': ['xoxo', 'blogroll']}* was observed in 736 malicious sites and 1,027 benign ones (461.34 malicious, 538.32 benign after attenuation) making it relatively more frequent in malicious sites. The feature *div{'id': 'content_disclaimer'}* was observed in no malicious sites and 62 benign ones (47.88 benign after attenuation) making it more frequent in benign sites. After manual inspection, we determined that this feature corresponded to a domain parking page where no other content was hosted on the domain.

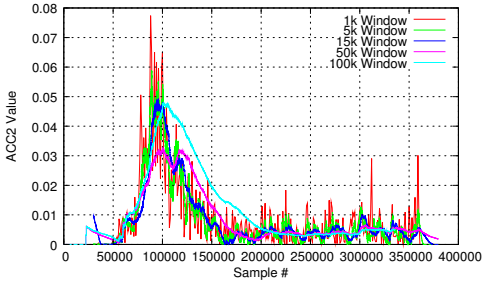


Figure 9: **Statistic value for *meta{'content': 'WordPress 3.3.1', 'name': 'generator'}* over time.** The statistic was computed over the experiment using window sizes of 1,000, 5,000, 15,000, 50,000 and 100,000 samples and uniform weighting.

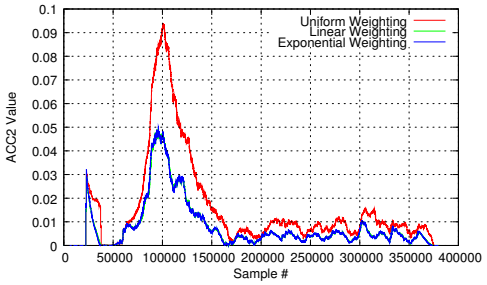


Figure 10: **Statistic value for *meta{'content': 'WordPress 3.3.1', 'name': 'generator'}* over time.** The statistic was computed over the experiment using a window size of 15,000 samples and various weighting techniques.

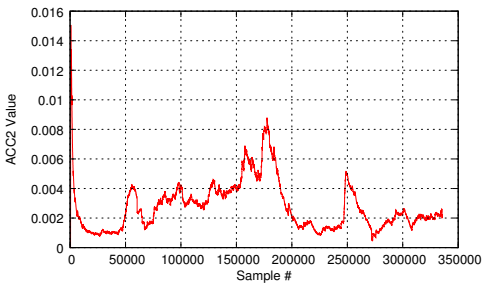


Figure 11: **Statistic value for *div{'id': 'content_disclaimer'}* over time.** The statistic was computed over the experiment using a window of size 15,000 samples and linear attenuation.

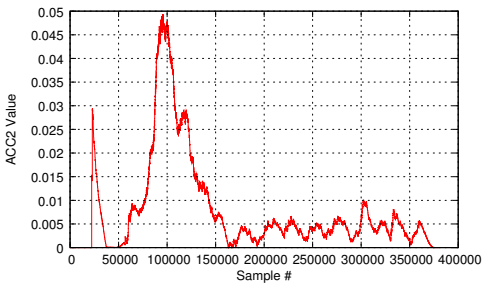


Figure 12: **Statistic value for *meta{'content': 'WordPress 3.3.1', 'name': 'generator'}* over time.** The statistic was computed over the experiment using a window of size 15,000 samples and linear attenuation.

The calculation of the ACC2 statistic for a feature at a particular time is parameterized by the window size and by a weighting scheme. As an example, Figure 9 shows the value of the statistic computed for the tag *meta{'content': 'WordPress 3.3.1', 'name': 'generator'}* over the experiment using different window sizes. When using a window, we compute the statistic by only considering examples that occurred within that window. We made passes over the data using window sizes of 1,000, 5,000, 15,000, 50,000 and 100,000 samples, which approximately correspond to 3 days, 2 weeks, 7 weeks, 24 weeks, and 48 weeks respectively.

A small window size generates a statistic value extremely sensitive to a few observations whereas a large window size yields a relatively insensitive statistic value. The window size thus yields a performance trade-off. If the statistic value for a feature is computed with a very small window, then the feature is prone to being incorrectly identified as meaningful, but will correctly be identified as meaningful with very low latency as only a few observations are needed. A large window will result in less errors regarding the usefulness of a feature but will create a higher latency.

Figure 10 shows the effect of varying the weighting scheme with a constant window size. Using a weighting scheme gives higher weight to more recent examples and the effect is very similar to simply decreasing the window size. There is almost no difference between exponential and linear decay.

Features belonging to positive (malicious) and negative (benign) examples often carry with them their own characteristics. The statistic values of negative examples tend to be relatively constant and time-invariant as the example in Figure 11 shows. These are generally features that indicate a lack of interesting content and therefore a lack of malicious content—for instance, domain parking pages. Conversely, the statistic value of positive examples tend to contain a large spike as evidenced by the example in Figure 12. The features correspond to vulnerable software and spike when an attack campaign exploiting that vulnerability is launched. Occasionally, additional spikes are observed, presumably corresponding to subsequent campaigns against unpatched software.

A design consideration when working with dynamic features is whether or not it is appropriate to use features that were highly ranked at some point in the past in addition to features that are currently highly ranked. As discussed above, negative features tend to be relatively constant and less affected, unlike positive features which fluctuate wildly. These positive features tend to indicate the presence of software with a known vulnerability that may continue to be exploited in the future.

Since it may happen that a feature will be useful in the future, as long as computational resources are available, better classification performance can be achieved by including past features in addition to the current top performing features. The result of including past features is that in situations where attack campaigns are launched against previously observed CMSs, the features useful for identifying such sites do not need to be learned again.

5.2 Classification performance

We ran the system with three different configurations to understand and evaluate the impact that different configurations had on overall performance. We send input to our ensemble of classifiers as "blocks," i.e., a set of websites to be used as examples. The first configuration generated content features from the very first block of the input stream but did not recompute them after that. The second configuration recomputed features from every block in the input stream but did not use past features which did not currently have a top statistic value. The third configuration used dynamic features in addition to all features that had been used in the past.

For all configurations, we used a block size of 10,000 examples for retraining the ensemble of C4.5 classifiers.

We also used a window size of 10,000 samples when computing the statistic value of features, and we relied on features with the top 100 statistic values. We generated ROC curves by oversampling the minority class by 100% and 200% and undersampling the majority class by 100%, 200%, 300%, and 500%. We ran each combination of over- and undersampling as its own experiment, resulting in a total of 10 experiments for each configuration. The true positive rate and false positive rate[2] for each experiment is taken as the average of the true positive and false positive rates for each block, that is, each block in the input stream to the system is tested on before being trained on, and the rates are taken as the average over the tested blocks.

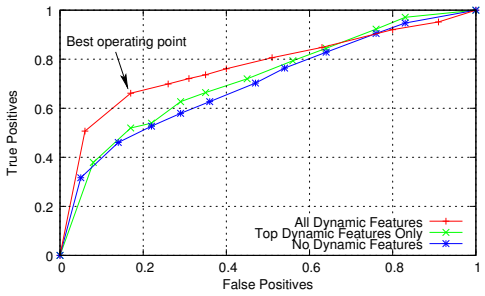


Figure 13: **ROC plot for three different strategies of dynamic features.** The classifier was run using three different configurations for dynamic features. The first configuration corresponds to classifiers trained on both current and past top features; the second corresponds to classifiers trained using only current top features; the third corresponds to classifiers trained using the top features from the first 5,000 samples.

Figure 13 shows the ROC curves generated for the three configurations described. The points resulting from the experiments have been linearly connected to form the curves. One can see that the configuration which used past features performed the best, followed by the configuration which used only current top dynamic features and the configuration which did not use dynamic features at all. The best operating point appears to achieve a true positive rate of 66% and a false positive rate of 17%.

The configuration which did not use dynamic features ended up selecting a feature set which was heavily biased by the contents of first block in the input data stream. While the features selected were useful on learning the first block, they did not generalize well to future examples since the distribution of pages that were observed had changed. This is a problem faced by all such systems in this setting that are deployed using a static set of features, unless the features set is fully expressive of the page content, i.e., all changes in the page content are able to be uniquely identified by a corresponding change in the feature values, then the features will eventually become less useful in classification as the distribution of pages changes.

The configuration which only used the current top dynamic features also performed relatively poorly. To understand why this is the case, we can see that in Figures 11 and 12 some features have a statistic value which oscillates to reflect the change in usefulness of the feature due to the time varying input distribution. One can also see that when a feature becomes useful, the corresponding increase in the statistic value lags behind since a few instances of the feature need to be observed before the statistic can obtain a high value again. During this transient period, the system fails to use features that would be useful in classification and so performance suffers. This problem may be partially addressed by shrinking the input block size from the data streams well as the window for computing the static value to a smaller value to reduce the transient. However such a strategy will still be outperformed by the strategy which remembers past features.

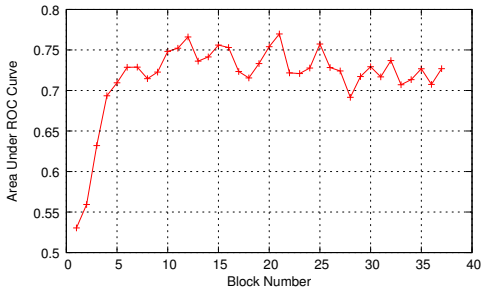


Figure 14: **AUC plot for the system over time using current and past dynamic features.** The system was run using both current and past top dynamic features. ROC curves were generated for each block of examples that was processed and the corresponding AUC value was computed.

For each input block in the experiments using past features, we recorded the true positive and false positive rates and used them to generate an ROC curve. We then used the ROC curve to approximate the area under the curve (AUC) which is a value that gives some intuitive understanding of how well the classifier performed on that block. Figure 14 shows the AUC values for each block in the experiment. The system performed relatively poorly until a sufficient number of blocks had been processed at which point the performance increased to a threshold value. We believe that the difficulty in achiev-

[2]The false negative rate and true negative rates are simple complements of the respective positive rates.

ing better performance is due to the nature of the problem, specifically it is not always the case that the content of a site and its traffic statistics are a factor in whether or not it will become compromised. We discuss this issue in more details in the limitations section.

Finally, we observed that when classification yielded a prediction that a site would become compromised, the reasoning can be read as the conjunction of conditions from the decision tree. For example the classification of `www.bisoft.org` which appeared in the search redirection data set was described as (Global Site Rank = 8) $\wedge$ (*<META NAME="Generator" CONTENT="EditPlus">= 1*) $\wedge$ (*<script type="text/javascript" src="/static/js/analytics.js">* = 1). The classification of benign examples was generally less obvious.

The conditions resulting in incorrect classification tended to follow a few main types. The first type are instances where a website would in the future become malicious, but very few examples like it exist at classification time. These sites contained content that would eventually yield prominent features for identifying sites that would become malicious but were not classified correctly due to latency in the dynamic feature extraction. As an example, consider in Figure 12 the samples that occurred just before the first significant spike.

The second type of instance that was incorrectly classified were examples that did not become malicious, but were classified as becoming so based on some strong positive content features that they contained. It is likely that after an initial attack campaign, vulnerable CMSs are less targeted due to the incremental number of compromises that could be yielded from them.

The third type of instance that was incorrectly classified were examples that would be become malicious for seemingly no apparent reason. These examples that would become malicious did not follow the general trend of large spikes corresponding to attack campaigns against a CMS, and have been observed with positive features both before and after its initial spike as well as with strong negative features. It is believed that these examples are cases where a site is becoming malicious for reasons completely independent of its content or traffic profile. It could be the case that an attack is launched where default login credentials for many CMSs are being attempted resulting in a few seemingly random breaks. It could also be the case that the domain in question was sold or rebuilt after observing it causing the system to erroneously predict its future malicious status from its old content.

6 Limitations

The limits on the classification performance of the system can be attributed to the following few difficulties in predicting if a site will become malicious.

Our system assumes the factors responsible for whether or not a site will become compromised can be summarized by its content and its traffic statistics. This assumption is sometimes violated, since for example sites may be compromised and become malicious due to weak administrator passwords being guessed or being retrieved via social engineering. Other examples may include adversaries who host their own sites with malicious intent. While it is often the case that such actors use similar page templates due to their participation in affiliate networks or out of convenience, such sites may introduce examples where the factors for the site being malicious are independent of its content. In such situations, the system will fail to perform well since the factors for site becoming malicious are outside its domain of inputs.

The nature of adversaries who compromise sites may also be perceived as a limitation on what our system can do. It has been observed that attack campaigns are launched where adversaries appear to enumerate and compromise sites containing a similar vulnerability. While adversaries do attack many sites which contain a particular vulnerability, it is generally not a reasonable assumption that they will systematically attack all sites containing this vulnerability both at the time of the campaign and in the future. The impact of this behavior on the system is that sites which contain similar content to those which were compromised in the campaign will be classified as becoming malicious in the future, when they actually may not since attackers have chosen to ignore them. While this does deteriorate the performance of the system, we argue that this does not take away its usefulness since these misclassifications represent sites which are still at considerable security risk and need attention.

The dynamic feature extraction system also presents at least two main limitations. The first is a correlation of features that are selected as top features at any given point in time. Tags often rise to the top of the list because they are part of some page template which has come up frequently. There may be multiple tags associated with a particular page template which all rise at the same time, and so a few of the top tags are redundant since they are identifying the same thing. It would be desirable to measure the correlation of the top features in order to select a more diverse and useful set however no attempt to do this was made in our experiments.

Another limitation of the dynamic features is that for system configurations which use past features in addition to the current top features, the size of the feature set is monotonically increasing. Thus, it will take longer over time train the classifiers and run the system. It would be

useful to further investigate the trade-offs between classification performance and limited feature lists.

Last, dynamic features introduce a unique opportunity for adversarial machine learning approach to poison the performance of the system. Adversaries which control a website may attempt to remove, change, or insert tags into their pages in order to damage the effectiveness of feature generation. For example, adversaries that host or control sites that have distinguishing tags may either try to remove them or rewrite them in semantically equivalent ways to prevent the system from using them for classification. Since the sites examined by the system are typically not under adversarial control at the time of evaluation, we believe that the impact of such attacks should be minimal; but it deserves further analysis.

7 Conclusions

We discussed a general approach for predicting a websites propensity to become malicious in the future. We described a set of desirable properties for any solution to this problem which are interpretability, efficiency, robustness to missing data, training errors, and class imbalance, as well as the ability to adapt to time changing concepts. We then introduced and adapted a number of techniques from the data mining and machine learning communities to help solve this problem, and demonstrated our solution using an implementation of these techniques. Our implementation illustrates that even with a modest dataset, decent performance can be achieved since we are able to operate with 66% true positives and only 17% false positives at a one-year horizon. We are currently working on making our software publicly available.

Acknowledgments

We thank our anonymous reviewers for feedback on an earlier revision of this manuscript, Brewster Kahle and the Internet Archive for their support and encouragement, and Jonathan Spring at CERT/SEI for providing us with historical blacklist data. This research was partially supported by the National Science Foundation under ITR award CCF-0424422 (TRUST) and SaTC award CNS-1223762; and by the Department of Homeland Security Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD), the Government of Australia and SPAWAR Systems Center Pacific via contract number N66001-13-C-0131. This paper represents the position of the authors and not that of the aforementioned agencies.

References

[1] Alexa Web Information Service. `http://aws.amazon.com/awis/`.

[2] DNS-BH: Malware domain blocklist. `http://www.malwaredomains.com/`.

[3] Norton safe web. `http://safeweb.norton.com`.

[4] Scrapy: An open source web scraping framework for Python. `http://scrapy.org`.

[5] Stop badware: A nonprofit organization that makes the Web safer through the prevention, remediation and mitigation of badware websites. `https://www.stopbadware.org/`.

[6] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proc. RAID'07*, pages 178–197, Gold Coast, Australia, 2007.

[7] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. NDSS'09*, San Diego, CA, February 2009.

[8] K. Borgolte, C. Kruegel, and G. Vigna. Delta: automatic identification of unknown web-based infection campaigns. In *Proc. ACM CCS'13*, pages 109–120, Berlin, Germany, November 2013.

[9] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM'99*, pages 126–134, New York, NY, March 1999.

[10] D. Chakrabarti, R. Kumar, and K. Punera. Page-level template detection via isotonic smoothing. In *Proc. WWW'07*, pages 61–70, Banff, Canada, May 2007.

[11] S. Debnath, P. Mitra, N. Pal, and C.L. Giles. Automatic identification of informative sections of web pages. *IEEE Transactions on Knowledge and Data Engineering*, 17(9):1233–1246, 2005.

[12] G. Forman. An extensive empirical study of feature selection metrics for text classification. *The Journal of machine learning research*, 3:1289–1305, 2003.

[13] J. Gao, W. Fan, J. Han, and P. Yu. A general framework for mining concept-drifting data streams with skewed distributions. In *Proc. SIAM SDM'07*, pages 3–14, Mineapolis, MN, April 2007.

[14] Google. Google Safe Browsing API. `https://code.google.com/apis/safebrowsing/`.

[15] L. Invernizzi, P. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna. Evilseed: A guided approach to finding malicious web pages. In *Proc. 2012 IEEE Symp. Sec. & Privacy*, pages 428–442, San Francisco, CA, May 2012.

[16] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proc. ACM CCS'11*, Chicago, IL, October 2011.

[17] J. John, F. Yu, Y. Xie, M. Abadi, and A. Krishnamurthy. deSEO: Combating search-result poisoning. In *Proc. USENIX Security'11*, San Francisco, CA, August 2011.

[18] L. Lancor and R. Workman. Using Google hacking to enhance defense strategies. *ACM SIGCSE Bulletin*, 39(1):491–495, 2007.

[19] N. Leontiadis, T. Moore, and N. Christin. A nearly four-year longitudinal study of search-engine poisoning. Tech. Rep. CyLab-14-008, Carnegie Mellon University. July 2014.

[20] N. Leontiadis, T. Moore, and N. Christin. Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In *Proc. USENIX Security'11*, San Francisco, CA, August 2011.

[21] K. Levchenko, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, A. Pitsillidis, N. Weaver, V. Paxson, G. Voelker, and S. Savage. Click trajectories: End-to-end analysis of the spam value chain. In *Proc. 2011 IEEE Symp. Sec. & Privacy*, Oakland, CA, May 2011.

[22] L. Lu, R. Perdisci, and W. Lee. SURF: Detecting and measuring search poisoning. In *Proc. ACM CCS 2011*, Chicago, IL, October 2011.

[23] MalwareBytes. hphosts online. `http://www.hosts-file.net/`.

[24] McAfee. Site Advisor. `http://www.siteadvisor.com/`.

[25] D. McCoy, A. Pitsillidis, G. Jordan, N. Weaver, C. Kreibich, B. Krebs, G. Voelker, S. Savage, and K. Levchenko. Pharmaleaks: Understanding the business of online pharmaceutical affiliate programs. In *Proc. USENIX Security'12*, Bellevue, WA, August 2012.

[26] J. P. McDermott. Attack net penetration testing. In *Proceedings of the 2000 workshop on New security paradigms - NSPW '00*, pages 15–21, New York, New York, USA, 2000.

[27] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall PTR, 2004.

[28] PhishTank. `https://www.phishtank.com/`.

[29] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your iFrames point to us. In *Proc. USENIX Security'08*, San Jose, CA, August 2008.

[30] Foster Provost and Tom Fawcett. Robust classification for imprecise environments. *Machine Learning*, 42(3):203–231, 2001.

[31] J. R. Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan Kaufmann, 1993.

[32] Ruihua Song, Haifeng Liu, Ji-Rong Wen, and Wei-Ying Ma. Learning block importance models for web pages. In *Proc. WWW'04*, pages 203–211, New York, NY, May 2004.

[33] Sophos. Security threat report 2013, 2013. `http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf`.

[34] The Internet Archive. Wayback machine. `https://archive.org/web/`.

[35] M. Vasek and T. Moore. Identifying Risk Factors for Webserver Compromise. In *Proc. Financial Crypto.'14*, Accra Beach, Barbados, February 2014.

[36] D. Wang, G. Voelker, and S. Savage. Juice: A longitudinal study of an SEO botnet. In *Proc. NDSS'13*, San Diego, CA, February 2013.

[37] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 01:69–101, 1996.

[38] L. Yi, B. Liu, and X. Li. Eliminating noisy information in web pages for data mining. In *Proc. ACM KDD'03*, pages 296–305, Washington, DC, August 2003.

[39] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proc. WWW'05*, pages 76–85, Chiba, Japan, May 2005.

Hulk: Eliciting Malicious Behavior in Browser Extensions

Alexandros Kapravelos[◇] *Chris Grier*[†*] *Neha Chachra*[‡] *Christopher Kruegel*[◇]
Giovanni Vigna[◇] *Vern Paxson*[†*]
[◇]*UC Santa Barbara* [†]*UC Berkeley* [‡]*UC San Diego*
[*]*International Computer Science Institute*
{*kapravel, chris, vigna*}*@cs.ucsb.edu* {*grier, vern*}*@cs.berkeley.edu* *nchachra@cs.ucsd.edu*

Abstract

We present Hulk, a dynamic analysis system that detects malicious behavior in browser extensions by monitoring their execution and corresponding network activity. Hulk elicits malicious behavior in extensions in two ways. First, Hulk leverages *HoneyPages*, which are dynamic pages that adapt to an extension's expectations in web page structure and content. Second, Hulk employs a *fuzzer* to drive the numerous event handlers that modern extensions heavily rely upon. We analyzed 48K extensions from the Chrome Web store, driving each with over 1M URLs. We identify a number of malicious extensions, including one with 5.5 million affected users, stressing the risks that extensions pose for today's web security ecosystem, and the need to further strengthen browser security to protect user data and privacy.

1 Introduction

All major web browsers today support broad extension ecosystems that allow third parties to install a wide range of modified behavior or additional functionality. Internet Explorer has binary add-ons (Browser Helper Objects), while Firefox, Chrome, Opera, and Safari support JavaScript-based extensions. Some browsers have online web stores to distribute extensions to users. For example, the most popular extension in Chrome's Web Store, AdBlock, has over 10 million users. Other popular extensions serve a variety of functions, such as preserving privacy, changing the aesthetics of the browser's UI, or integrating with web services such as Google Translate.

The amount of critical and private data that web browsers mediate continues to increase, and naturally this data has become a target for criminals. In addition, the web's advertising ecosystem offers opportunities to profit by manipulating a user's everyday browsing behavior. As a result, malicious browser extensions have become a new threat, as criminals realize the potential to monetize a victim's web browsing session and readily access web-related content and private data.

Our work examines extensions for Google Chrome that are designed with malicious intent—a threat distinct from that posed by attackers exploiting bugs in benign extensions, which has seen prior study [6, 5]. Extensions for Google Chrome are primarily distributed through the Chrome Web Store.[1] Like app stores for other platforms, such as Android or iOS, inherent risks arise when downloading and executing programs from untrusted sources. Reports have documented not only malicious extensions [27], but miscreants *purchasing* extensions (and thereby access to their userbases via update mechanisms) to add malicious functionality [2, 25]. In addition to the web store, extensions can also be directly installed by users and other programs. Installed by a process called *sideloading*, these extensions pose a recognized risk that browser vendors have attempted to prevent through modifications to the browser [22]. Sideloaded extensions are especially problematic since they can be installed without user knowledge, and are not subject to review by a web store. Despite efforts to stifle sideloaded extensions, they remain a significant problem [12].

In this paper we present Hulk, a tool for detecting malicious behavior in Google Chrome extensions. Hulk relies on dynamic execution of extensions and uses several techniques to trigger malicious functionality during execution. One technique we developed to elicit malicious behavior is the use of *HoneyPages*: specially-crafted web pages designed to satisfy the structural conditions that trigger a given extension. We interpose on all queries and modifications to the DOM tree of the HoneyPage to automatically create elements and mimic DOM tree structures for extensions on the fly. Using this technique, we can readily observe malicious behavior that inserts new `iframe` or `div` elements.

In addition, we built a fuzzer to drive the execution

[1] https://chrome.google.com/webstore/category/extensions

of event handlers registered by extensions. In our experiments, we use the fuzzer to trigger all event handlers associated with web requests, exercising each with 1 million URLs. Although we undertook extensive efforts to trigger malicious behavior, the possibility remains that Hulk lacks the mechanisms to satisfy all of the conditions necessary for eliciting an extension's malicious behavior.

Our analysis of 48,332 Chrome extensions found that malicious extensions pose a serious threat to users. By developing a set of rules that label execution logs from Hulk, we identified 130 malicious extensions and 4,712 "suspicious" extensions, most of which appear in the Chrome Web Store. Several large classes of malicious behavior appear within our set of extensions: affiliate fraud, credential theft, ad injection or replacement, and social network abuse. In one case, an extension performing ad replacement had nearly 2 million users, similar in size to some of the largest botnets.

In summary, we frame our contributions as follows:

- We present Hulk, a system to perform dynamic analysis for Chrome extensions.
- We demonstrate the effectiveness of HoneyPages and event handler fuzzing to elicit malicious behavior in browser extensions.
- We perform the first broad study of malicious Chrome extensions.
- We characterize several classes of malicious Chrome extensions, some with very large footprints (up to 5.5M installations) and propose solutions to eliminate entire classes of malicious behavior.

2 Background

We begin by reviewing the Google Chrome extension model and the opportunities this model provides to malicious extensions.

2.1 Chrome Extension Composition

Google Chrome supports extensions written in JavaScript and HTML (distributed as a single `zip` file). A small number of extensions also include binary code plugins, although these are subject to a manual security review process [15]. Each extension contains a (mandatory) manifest that, along with other extension parameters, describes the permissions the extension uses and the list of resources that the browser should load.

The permission system is designed in the spirit of least privilege, with the goal of limiting the resources available to an extension in case it has exploitable vulnerabilities [5]. The threat model does not attempt to address malicious extensions accessing sensitive content or performing other actions. The permission system determines which sites an extension can access, the allowed API calls, and the use of binary plugins. We describe relevant parts of the permission system later in this section. See Barth et al. for a more detailed description of Chrome's extension architecture [5].

2.2 Installing Extensions

The Chrome Web Store is the official means for users to find and install extensions. The web store is similar to other app stores, such as those for iOS and Android, in that developers create extensions and upload them to the store for users to download. Extension developers can also push out updates without requiring any action by the end-user.

In addition to the Chrome Web Store, extensions can also be installed manually by a user or an external program. We refer to the installation of extensions outside the web store as *sideloading*. Chrome version 25 (released February, 2013) included changes to prevent silent installation of Chrome extensions and require that the user indicate consent for installation [22]. In May, 2014, Chrome took further steps to prevent sideloading by requiring *all* installed extensions to be hosted in the Chrome Web Store [18]. While these changes increase the difficulty of sideloading, it is still possible for programs to force silent installation of extensions, since the attacker already has control of the machine. For our study we obtained a set of extensions that are sideloaded into Chrome by other Windows programs, many of which are known malware.

2.3 Extension Permissions

Permissions. Chrome requires extensions to list the permissions needed to access the different parts of the extension API. For example, Figure 1 shows a portion of a manifest file requesting permission to access the `webRequest` and `cookies` API. The `webRequest` permission allows the extension to "observe and analyze traffic and to intercept, block, or modify requests in-flight" by allowing the extension to register callbacks associated with different parts of the HTTP stack [15]. Similarly, the cookies API allows the extension to get, set, and be notified of changes to cookies.

The extension API permissions operate in conjunction with the optional *host permissions*, which limit the API permissions to access resources only for the specified URLs. For example, in Figure 1 the extension requests host permissions for `https://www.google.com/`, which allows it to access `cookies` and `webRequest` APIs for the specified domains. Host permissions also support wildcarding

```
...
"permissions": [
  "cookies",
  "webRequest",
  "*://*.facebook.com/",
  "https://www.google.com/"
],
...
"content_scripts": [
  {
    "matches": ["http://www.yahoo.com/*"],
    "js": ["jquery.js", "myscript.js"]
  }
],
...
"background": {
  "scripts": ["background.js"]
},
...
"content_security_policy": "script-src 'self'
      http://www.foo.com 'unsafe-eval';"
...
```

Figure 1: Example of a manifest that shows API permissions for two hosts, followed by content scripts that run on `http://www.yahoo.com`, followed by a background script that runs on all pages. Finally, the CSP specifies the ability to include and `eval` scripts in the extension from `foo.com`.

URLs. In Figure 1, the extension requests access to `*://*.facebook.com`. This permission allows for access to all subdomains of `facebook.com` requested via any URL scheme. In addition to wildcards, the special token `<all_urls>` matches any URL.

Besides the permissions described above, we found that extensions request a variety of other permissions. In Section 4 we summarize the permissions requested for all of the extensions we examined, and we discuss the permissions relevant to various types of abuse in Section 5. Other resources provide a thorough analysis of the Chrome permission system [5, 6].

Content Scripts. In addition to permissions for accessing various resources associated with a page, extensions can also specify a list of `content_scripts` to indicate JavaScript files that will run inside of the web page. Figure 1 shows an example of including two JavaScript files, `jquery.js` and `myscript.js` that will be run in the context of the page for any URLs matching the specified URL patterns (all pages on `http://www.yahoo.com/` in this example). Inside of each JavaScript file the author can include further logic to decide if and when to execute.

The ability to run in the context of a page is a powerful feature. Once a content script executes, any resulting actions become indistinguishable from actions performed by JavaScript provided by the web server. Not only can the scripts modify the DOM tree or other scripts, but they can also issue authenticated web requests (such as POST with proper cookies).

Background Pages. Besides the content scripts that allow an extension to interact with a given page, Chrome also allows extensions to run scripts in a "background page". Figure 1 shows an example manifest file that specifies `background.js` as a background page. Background pages often contain the logic and state an extension needs for the entirety of the browser session and do not have any visibility to the user. For example, an extension requesting `webRequest` permissions may use the background script to attach a listener to read outgoing requests using the `chrome.webRequest.onBeforeRequest.addListener()` call. After filtering on the *host permissions*, Chrome will send the extension a notification for every outgoing request. We detail further examples in the context of the extensions in the following sections.

Content Security Policy. In general, servers can specify a Content Security Policy (CSP) header that the browser uses to determine the sources from which it can include objects on the page. CSP can also specify other options, such as whether to allow the page to perform an `eval` or to embed inline JavaScript [29]. Extensions can use the same syntax to express their CSP in the manifest file. For example, an extension that wishes to include source from `foo.com` and to execute `eval` can specify its CSP as shown in Figure 1.

3 Architecture

In this section, we describe the architecture of Hulk, our dynamic analysis system that identifies malicious behavior in Chrome extensions. Hulk dynamically loads extensions in a monitored environment and observes the interaction of extensions with the loaded web pages. Using a set of heuristics to identify potentially dangerous behavior, it labels extensions as *malicious*, *suspicious*, or *benign*. In the rest of this section we describe how Hulk works and the challenges that arise in analyzing browser extensions.

3.1 Profiling Extensions

At the core of our dynamic analysis system is an instrumented browser and extension loader that enables us to automatically install extensions and instrument activity during web browsing. Our monitoring hooks collect data

from multiple vantage points within Hulk as it visits web pages and triggers a range of extension behavior.

URL Extraction. Before we dynamically analyze an extension we need to ensure that we can trigger the extension's functionality. Most extensions interact with the content of web pages, so we need to choose which URLs to load for our analysis. To this end, we use three sources of URLs: the manifest, the source code, and a list of popular sites. First, using the manifest file of the extension we construct valid URLs that match the permissions and content scripts specified. In some cases, the host permissions of an extension are restrictive—for example, `https://*.facebook.com`—so we can generate URLs that will match the pattern. It is more difficult to pick URLs to visit in cases where the extension requests host permissions on all URLs (Section 2.3), because the malicious behavior may only trigger on a small subset of sites. Therefore, we search the source code for any static URLs and visit those as well. Finally, for every extension we also visit a set of popular sites targeted by malicious extensions. We constantly strive to improve this list as we detect malicious extensions attacking particular domains. We however note that although we use multiple sources of URLs to determine the appropriate pages to visit, our approach is not complete; we discuss the limitations further in Section 7.

HoneyPages. Some extensions activate based on the content of a web page instead of the URL. To analyze such extensions we use specially crafted pages that attempt to satisfy the conditions that an extension looks for on a page before performing an action. We call these *HoneyPages*. HoneyPages contain JavaScript functions that overload built-in functions that query the DOM tree of the web page. As a result, when an extension queries for the presence of a specific element we can automatically create it and insert it into the page. For example, if the extension queries an `iframe` DOM element with the intention to alter it, then our HoneyPage will create an `iframe` element, inject it in the DOM tree, and return it to the extension.

HoneyPages enable us to supplement the URL extraction phase and dynamically create an environment for the extension to perform as many actions as it needs. The on-demand nature of a HoneyPage does not restrict us to a specific DOM tree structure, but enables us to determine what an extension looks for in a page during execution, since we can record all interactions within a HoneyPage. By using HoneyPages we can better understand how the extension will behave on arbitrary pages that are otherwise difficult to generate prior to analysis.

3.2 Event-Based Execution

The Chrome browser offers to extensions an event-based model to register callbacks that respond to certain browser-level events. For example, extensions use the `chrome.webRequest.onBeforeRequest` callback to intercept all outgoing HTTP requests from the browser. HoneyPages will not trigger callbacks for network events that require special properties, such as a specific URL or HTTP header. Therefore, we complement HoneyPages with event handler fuzzing. Specifically, we invoke all event callbacks that an extension registers in the `chrome.webRequest` API with mock event objects. We point to a HoneyPage loaded in the active tab while invoking the callbacks, enabling us to monitor the changes that the extension attempts to make on that page. Our approach allows us to test for every extension the extension's callbacks on the top 1 million Alexa domains in under 10 seconds on average.

3.2.1 Monitoring Hooks

Browser Extension API. Depending on the permissions included in the manifest (Section 2.3), an extension can use the Chrome extension API to perform actions not available to JavaScript running in a web page. As such, monitoring the extension API captures a subset of the total JavaScript activity that results from an extension, but gives us a detailed picture of what the extension attempts to do. For example, we monitor the extension API and log if the extension registers a callback to intercept all HTTP requests performed by the browser, and then track the changes that the extension makes to the HTTP requests. To do this, we leverage the current logging infrastructure offered by Chrome for monitoring the activity of extensions. We build upon the JavaScript function call logging provided by the browser to identify malicious behavior, such as tampering of security-related HTTP headers.

Content Scripts. We intercept and log all additional code introduced by the extension in the context of the visited page. Doing so provides a more complete picture of the extension's functionality, since it can include remote scripts from arbitrary locations and inject them into the page. Remote scripts can compromise the page's security similar to third-party JavaScript libraries [23], and make the analysis of the extension more difficult. Using remote scripts gives miscreants the ability to blacklist IP addresses of our analysis system (i.e., cloaking [17, 28]) or return code without the malicious components. Remote JavaScript inclusion also renders static analysis on the extension's code fundamentally incomplete since parts of the extension's codebase are not available until execution.

Network Logging. We use a transparent proxy that intercepts all browser HTTP and DNS traffic to log the requests made during extension execution. A browser extension has a set of files available as resources loaded by the browser, and it can also download and execute content from the web. Since the URLs retrieved can be computed at runtime, monitoring the network activity of the extension is critical for a complete analysis of its source code and included components. In addition to identifying remote content, we log all domains contacted by monitoring the DNS requests generated by the browser. Doing so enables us to identify extensions that contact nonexistent domains, which can occur because the extension is no longer operational or up-to-date. In these cases, our analysis was necessarily incomplete, since when the domain was active the extension could have fetched more remote code from it.

3.3 Detecting Malicious Behavior

As described in the previous section, our dynamic analysis system can provide detailed information about all browser and extension activity performed while visiting web pages. We combine this data to label the extension as either *benign*, *suspicious*, or *malicious* by applying a set of labeling heuristics based on the behavior. Labeling an extension as malicious indicates we identified behavior harmful to the user. Suspicious indicates the presence of potentially harmful actions or exposing the user to new risks, but without certainty that these represent malicious actions. Finally, when we do not find any suspicious activity, we label the extension as benign.

3.3.1 JavaScript Attributes

We use our monitoring modules described in Section 3.2.1 to identify malicious JavaScript execution. Below we detail actions that we consider malicious or suspicious in our post-processing analysis.

Extension API. As described earlier, Chrome's extension API offers privileged access to additional functionality of the browser besides native JavaScript, using permissions specified in the manifest file. While there are benign uses for every permission, we found several extensions that abuse the API. Specifically, for reasons described below, we consider the following actions available only through the extension API as malicious: uninstalling other extensions, preventing uninstallation of the current extension, and manipulating HTTP headers.

We consider uninstalling other extensions as malicious because some extensions uninstall cleaner extensions, such as the extension Facebook created to remove harmful extensions on its blacklist.[2] We detect this behavior by monitoring the `chrome.management.uninstall` API calls. To avoid false positives, we can differentiate cleaners from malicious extensions because, to the best of our knowledge, cleaners operate in a different fashion than Antivirus does: they clean up malicious extensions and then remove themselves from the browser. This differs from the behavior of malicious extensions, which remain persistent on the system.

Besides attempting to uninstall other extensions, malicious extensions often prevent the user from uninstalling the extension itself. More specifically, we found extensions that prevent the user from opening Chrome's extension configuration page where a user can conveniently uninstall any extension. To prevent uninstallation, malicious extensions interfere with tabs that point to the extension configuration page, `chrome://extensions`, either by replacing the URL with a different one, or by removing the tab completely. For analysis, we load a tab with `chrome://extensions` in the browser during our dynamic analysis and monitor any interactions to identify such behavior.

Lastly, using callbacks in the `webRequest` API, a malicious extension can manipulate HTTP headers. Extensions can use the `webRequest` API to effectively perform a man-in-the-middle attack on HTTP requests and responses before they are handled by the browser. This behavior is often malicious (or at least dangerous) since we found extensions that remove security-related headers, such as Content-Security-Policy or X-Frame-Options, through the use of callbacks such as `webRequest.onHeadersReceived` and `webRequestInterval.eventHandled`. By monitoring the use of this API, we can log events that reveal state of HTTP headers before and after the request. Upon manipulation of any security-related headers, we label the extension as malicious.

Interaction with visited pages. In addition to the extension API, we also monitor an extension's use of content scripts to modify web content loaded in the browser. In our analysis, we flag two kinds of interaction: sensitive information theft as malicious and injection of remote JavaScript content as suspicious.

There are many ways an extension can steal personal information from the user. For example, it can act as a JavaScript-based keylogger by intercepting all keystrokes on a page. Extensions can also access form data, such as a password field, before it is encrypted and sent over the network. Finally, extensions can also steal sensitive information from third parties by accessing sites with which the user has a valid session, and ei-

[2] `https://chrome.google.com/webstore/detail/facebook-malicious-extens/mhkafblddkepdhhjpmedkngigkjjknoa`

ther issuing requests to exfiltrate data, or simply stealing valid authentication tokens.

We label any extension that injects remote JavaScript content into a web page as suspicious. We define this activity as adding a `script` element with a `src` attribute pointing to a domain that is different from the one of the web page. Including these scripts complicates analysis since the JavaScript content can change without any corresponding change in the extension. We have observed changes to JavaScript files that substantially alter the functionality of an extension, possibly due to a server compromise.

3.3.2 Network Level

By monitoring network requests, including DNS lookups and HTTP requests, we identify other types of suspicious/malicious behavior. Using a manual analysis of network logs we have identified two attributes that indicate malicious or suspicious behavior: request errors and modification of HTTP requests. To detect HTTP modifications, we examine if the network response that we observe on the wire differs from the network response finally processed by the browser.

As we discussed earlier, the extension API offers callbacks to give extensions the ability to intercept and manipulate web requests. Not only can extensions drop security-related headers, but extensions can change or add parameters in URLs before the HTTP request is sent. We find such suspicious behavior common, especially among extensions that request permissions on shopping-related sites such as Amazon, EBay, and others. In these cases, the extension adds parameters to the URL that indicate that the site should credit a particular affiliate for any resulting sales. We discuss this behavior in more detail in Section 5. At the network level, we have the complete view of how the requests originally appeared. We combine that knowledge with our `chrome.*` API monitoring to identify the exact changes made to the request.

We also look for errors during domain name resolution to identify extensions that contact domains since taken down. As with drive-by downloads, we expect that malicious code dynamically loaded into an extension will eventually become blacklisted. In such cases, the extension will fail to introduce more code during its execution. We detect this behavior and mark it as suspicious.

3.4 Injected Content Analysis

A Chrome extension can also manipulate the visited pages of the browser by injecting a content script. The injected script runs in the context of the visited page and thus has full access to its DOM tree. The injected code can vary significantly, and, with the dynamic nature of JavaScript, can prove difficult to analyze statically. The use of HoneyPages enables us to understand the injected code's full intentions. Instead of trying to infer what the code will do, we actually run it to observe its effects on the DOM tree and classify it accordingly. For example, if the injected code looks for a form field with the name "password," we classify it as malicious, since it can potentially hijack the user's credentials on the page. Another example concerns injecting additional code, where the injected code is part of a two-stage process that fetches yet more code from the web and dynamically executes it in the context of the visited page. By relying on HoneyPages to understand the code's intentions by the effect that the code has on a given page, we obtain a more precise view of what the code attempts do than we can using only static analysis.

Analysis result	Count
Malicious	130
Suspicious	4,712
Benign	43,490
Total	48,332

Table 1: Classification distribution of extensions.

Detection class	Count
[s] Injects dynamic JavaScript	2,672
[s] Produces HTTP 4xx errors	2,322
[s] Evals with input >128 chars long	451
[m] Prevents extension uninstall	56
[m] Steals password from form	39
[s] Performs requests to non-existent domain	26
[m] Contains keylogging functionality	23
[m] Injects security-related HTTP header	11
[m] Steals email address from form	10
[m] Uninstalls extensions	8

Table 2: Distribution of detected suspicious/malicious behavior from analyzed extensions. Notice that an extension might have more than one detections and that we mark with [m] detections classified as malicious and with [s] detections classified as suspicious.

4 Results

To evaluate Hulk we use two sources of extensions: the official Chrome Web Store (totaling 47,940 extensions), and extensions sideloaded by binaries. We obtained the latter based on binaries executed in Anubis [1], which, after removing a large number of duplicates, resulted in

Rank	Top 10 types of permissions	# ext.
1	tabs	16,787
2	notifications	12,011
3	unlimitedStorage	9,424
4	storage	5,725
5	contextMenus	4,774
6	cookies	2,872
7	webRequest	2,849
8	webRequestBlocking	2,102
9	webNavigation	1,623
10	management	1,533

Table 3: The top 10 permissions found in the manifest files for all extensions we ran. Extensions can include more than one permission.

a set of 392 unique extensions. As shown in Table 1, in total we analyzed 48,332 distinct extensions, of which Hulk labeled 130 as *malicious* and 4,712 as *suspicious*. Table 2 summarizes all of the detected behaviors, which we analyze in more detail in the following sections.

4.1 Permissions Used

In this section we characterize the extensions we executed by identifying the most popular permissions, content scripts, and API calls that they performed.

Permissions. Table 3 shows the top 10 permissions from 30,392 unique extensions that use the Chrome Extension API (excluding the host permissions). The most commonly used, the `tabs` permission, allows an extension to interact with the browser's tabs, including navigating a tab to a specified URL and registering callbacks to react to changes in the address bar. The second most popular permission, `notifications`, allows an extension to generate custom notifications that alert the user. The `storage` and `unlimitedStorage` permissions allow storing of permanent data in the user's browser. The `contextMenus` permission allows an extension to add additional items on the context menu of the browser. Context menus appear when the user right clicks on a page. To manipulate the browser's cookies, an extension needs to ask for the `cookies` permission. The permissions `webRequest`, `webRequestBlocking` and `webNavigation` allow an extension to inspect, intercept, block, or modify web requests from the browser. Finally, an extension can get a list of other extensions installed in the browser—and even disable or unistall them—with the `management` permission.

We also computed permission statistics independently for the set of benign extensions and the set of malicious or suspicious ones. To our surprise, we found that permissions for benign extensions do not differ significantly from permissions requested by malicious/suspicious ones, indicating that often attackers do not need to target different APIs to perform their attacks; maliciousness instead manifests in the way they use the API.

Rank	Top 25 hosts in permissions	# ext.
1	http://*/*	7,319
2	https://*/*	6,395
3	<all_urls >	2,044
4	http://*/	1,126
5	*://*/*	1,025
6	https://*/	665
7	www.flashgame90.com/Default.aspx	224
8	https://api.twitter.com/	200
9	http://localhost/*	161
10	http://127.0.0.1/*	133
11	https://secure.flickr.com/	95
12	*://*.facebook.com/*	91
13	*://*/	89
14	https://www.facebook.com/*	82
15	http://vk.com/*	77
16	http://*.facebook.com/*	77
17	https://mail.google.com/*	71
18	https://*.facebook.com/*	70
19	http://*.google.com/	68
20	https://www.google-analytics.com/	67
21	https://mail.google.com/	64
22	https://*.google.com/	62
23	https://twitter.com/*	61
24	https://www.googleapis.com/	60
25	google.com/accounts/OAuthGetAcc[..]	56

Table 4: The top 25 host permissions used by extensions. Extensions can include more than one host permission per manifest.

We found 18,313 extensions that use host permissions to restrict on which pages the extension can use the privileged `chrome.*` API. Table 4 shows the top 25 hosts appearing in host permissions. As seen in the table, extensions typically request broad permissions using wildcards in URL patterns. In addition these, we examined the hosts that extensions specified as targets for injecting content scripts, per Table 5, finding similar broad declarations. In practice, extension authors often use content scripts and host permissions in an unrestricted fashion.

API calls. Table 6 shows the top 15 Chrome Extension API calls made during by extensions during our experiments. There are several measurement artifacts introduced by our methodology. To load an extension for testing, we install the extension on a clean

Rank	Top 25 hosts in content_scripts	# ext.
1	http://*/*	12,472
2	https://*/*	10,864
3	<all_urls>	4,795
4	*://*/*	1,536
5	https://www.facebook.com/*	520
6	*://*.facebook.com/*	510
7	https://mail.google.com/*	458
8	http://www.facebook.com/*	433
9	https://*.facebook.com/*	344
10	http://*.facebook.com/*	320
11	file://*/*	315
12	https://twitter.com/*	303
13	http://mail.google.com/*	273
14	*://pages.brandthunder.com/[..]	265
15	https://plus.google.com/*	261
16	ftp://*/*	246
17	http://vk.com/*	227
18	http://www.youtube.com/*	211
19	file:///*	207
20	*://mail.google.com/*	189
21	http://twitter.com/*	179
22	*://www.facebook.com/*	178
23	http://ak.imgfarm.com/images[..]	177
24	*://*.reddit.com/*	164
25	https://vk.com/*	164

Table 5: The top 25 hosts used in extensions' content script permissions.

browser each time we start an analysis. This causes `runtime.onInstalled` to appear in every analysis independent of the extension's activities. We also open the `chrome://extensions` tab from inside the extension to determine if the extension interferes with the management of extensions. This causes Hulk to record a large number of `tabs.create` calls. In Table 6 the `tabs` API is by far the most used API, which matches the popularity of `tabs` permissions observed in Table 3.

4.2 Network Level

Using network activity alone we identified 24 malicious extensions. These extensions were labeled as malicious by Hulk because they tampered with security-related HTTP headers. By removing HTTP response headers like *Content-Security-Policy*, the malicious extensions can inject JavaScript into pages that specifically do not allow scripts from external sources (according to the CSP policies provided by the web server). For example, Hulk found multiple variants of an active extension on the Chrome Web Store targeting users that seek to cheat in online games; these extensions, generally going by the name "*Cheat in your favorite games*", affect over 20K users.

Rank	Top 15 chrome.* APIs called	# calls
1	runtime.onInstalled	182,476
2	webRequestInternal.eventHandled	57,466
3	tabs.getAllInWindow	49,312
4	tabs.onUpdated	32,354
5	tabs.create	25,947
6	i18n.getMessage	13,549
7	webRequest.onBeforeSendHeaders	13,213
8	runtime.connect	13,004
9	extension.getURL	11,942
10	storage.get	10,178
11	contextMenus.create	7,816
12	tabs.get	6,970
13	webRequest.onBeforeRequest	6,168
14	runtime.sendMessage	5,847
15	extension.sendRequest	5,454

Table 6: The top 15 chrome.* APIs called by extensions during dynamic analysis.

During our experiments we encountered cases where our analysis could not obtain the full set of information needed to make a decision regarding the maliciousness of an analyzed extension. This problem arose due extensions performing HTTP requests that either returned errors, such as an HTTP 404 responses, or having domain names that no longer resolved. In such cases, given our inability to exercise the extension's full set of capabilities, and because the failed requests might correspond to fetching additional code, we mark these extensions as suspicious.

4.3 Extensions Management

Using signals tailored to detect the manipulation of the `chrome://extensions` page (as described in Section 3.3), we found several extensions on the Chrome Web Store that prevent uninstallation. Two of of these extensions claim to be video players (each with thousands of user) and completely replace Chrome's extensions management with a page that prevents users from uninstalling them. These are "HD Video Player" with 7,173 users and "SmartScreen Video Plugin" with 11,012 users. These signals also generated a false positive: the "No Tab Left Behind" extension (with only 8 users) allows only one tab at a time to be open. Thus, during our execution this extension prevented us from opening the extension settings tab.

4.4 Code Injection

Code injection was the most commonly detected "suspicious" feature in our dataset. In principle injection need not occur at all, since Chrome extensions can come packaged with all the code needed to operate. In total, we found more than 3,000 extensions that dynamically introduced remotely-retrieved code either through script injections or by evoking `eval`. As we noted earlier, using remote code renders static analysis on the extension's code fundamentally incomplete. However, Hulk can identify code injections and pinpoint the remote locations from which an extension fetches code. Although not necessarily malicious, we found many cases of dangerous code injection. For example, our system identified an extension named "Bang5TaoShopping assistant" from the Chrome Web Store that has been installed in 5.6 million (!) browsers and injects code into every visited page. Several extensions perform this same activity, while others insert tracking pixels for similar purposes. One instance sends cleartext HTTP request to a server controlled by the extension that encodes the URL visited by the user along with a unique identifier, leaking users browsing behavior and thus compromising their privacy.

5 Profiting from Maliciousness

In this section, we discuss five categories of malicious behavior in extensions, and describe their characteristics and the methods they employ to carry out their goals. We base each of these categories on examples we found in our feeds. When the extension is available on the Chrome Web Store, we also when possible include the number of users prior to reporting the extension to Google for review.

We have reported to Google any extension that performs behavior that is clearly abusive or malicious, and several of our reports have lead to removals of extensions from the web store.

5.1 Ad Manipulation

Advertisement manipulation falls in a grey area in that it does not subvert the user, but rather manipulates an external ecosystem. Replacing ads might appear benign to end users, but removes the potential for monetary credit for website owners (publishers) and instead fraudulently credits the extension owner. We include in this category the addition of new ads as well as the replacement of existing ads or identifiers. We find a range of behaviors in extensions, such as replacing banner ads with different identically-sized banners; inserting banners and text ads into well-known sites (such as Wikipedia); changing affiliate IDs for ads; or simply overlaying ads on top of

```
"content_scripts": [{
  "matches": ["http://*/*", "https://*/*"],
  "js": ["js/content.js"]
}],
"permissions": ["http://*/*",
  "https://*/*", "tabs"],
```

Figure 2: Permission-related JSON from the manifest file of an extension performing ad replacement.

content. Each instance aims to profit from impressions or clicks on the substituted advertisements.

As one striking example of ad manipulation we found an extension on the Chrome Web Store that had 1.8M users at the time we detected it. The extension, named "SimilarSites Pro" used primarily unobfuscated JS to perform benign functionality as advertised on the Chrome Web Store; however, it also inserted a script element into the content of web pages that downloads another, fully-obfuscated script (using `eval` and `unescape`) from a web server. At the time of analysis, this script contained a large conditional block that looked for `iframe` elements of particular sizes, such as 728x90 pixels, and replaced them with new banners of the same size. Since our first analysis, we have seen several new versions of the script available from the same URL. In addition, the extension contains a blacklist of sites and meta keywords where it should *not* change the banners, which appears due to many ad networks prohibiting the display of their ads on porn sites.

We find the same JavaScript included in five other extensions from the Chrome Web Store, as well as one sideloaded extension. Based on manual analysis, these extensions are primarily produced by a single company called "SimilarGroup" that engages in dubious behavior through the Chrome Web Store.

To perform banner replacement, the extension requests the permissions shown in Figure 2. Such exceptionally wide permissions are not uncommon [6]. Therefore, their presence alone provides little insight into the functionality of the extension. The most significant permission in Figure 2 is the broad use of content scripts that allow the extension to inject dynamic JavaScript files from a remote location. Following injection, execution continues as though the page had included it. Such content scripts provide an exceptionally powerful feature to enable a variety of malicious behaviors, as further discussed in this Section.

5.2 Affiliate Fraud

Many major merchant web sites such as `amazon.com`, `godaddy.com`, and `ebay.com` run affiliate programs that

credit affiliates with a fraction of the sales made as a result of customers referred by the affiliates. Usually merchant programs assign unique identifiers to affiliates, which affiliates then include in the URL that refers customers to the merchant site. Furthermore, affiliate programs usually associate a cookie with the user's browser so that they can attribute a sale to an affiliate within several hours after a user originally visited the merchant site with an affiliate identifier.

As an example, when a user reads product reviews on an Amazon affiliate's blog and clicks on a link to Amazon, the link includes an Amazon affiliate ID specified with the `tag` parameter in the URL, such as `http://www.amazon.com/dp/0961825170/?tag=affiliateID`. When Amazon receives this request, it returns a `Set-Cookie` header with a cookie that associates the user with the affiliate. When the customer returns to Amazon within 24 hours and makes a purchase, Amazon credits the affiliate with a small percentage of the transaction amount.

Such programs expect affiliates to bring potential customers to their sites via affiliate pages that advertise the merchant products. However, we found examples of several extensions involved in *cookie stuffing*—a technique that causes the user's browser to visit the merchant URLs without the user clicking on affiliate URLs. Doing so causes the merchant to deliver a cookie associated with the fraudulent affiliate, who then receives credit for any future, unrelated purchase made by the customer on the merchant site. Besides defrauding the merchant, the fraudulent affiliate also causes an over-write of the cookie associated with any legitimate affiliate who might have genuinely influenced the user to buy the product.

In our study, we found two kinds of extensions that defrauded affiliate programs. The first group includes extensions that provide some utility to users—such as refreshing pages automatically every few seconds, or changing the theme of popular sites like Facebook—but do not inform users of the extension author profiting from the user's web browsing. Generally, these activities involve monitoring visited URLs for merchant sites where the extension can earn a commission and modifying the outgoing requested URLs to include the affiliate ID, or by injecting `iframe`'s that include affiliate URLs.

For example, we found an extension named "*Split Screen*" (with 52K users) that allows users to show two tabs in a single window, while also stealthily monitoring the URLs visited by the user. It then silently replaces the requested URL with the affiliate's URL for sites such as `amazon.com`, `amazon.co.uk`, `hotelscombing.com`, `hostgator.com`, `godaddy.com`, and `booking.com`. For some merchants, it also sets the referrer header for outgoing requests to falsely imply a visit through the affiliate's site. The extension is able to make these changes using `tab` and `webRequest` permissions, as well as by registering callbacks on `chrome.tabs.onUpdated` to identify changes in the URL as a user types, and `chrome.webRequest.onBeforeSendHeaders` to modify the referrer header before the browser sends a request to a merchant site. We found four other extensions created by the same developer that similarly provided some small utility to the user while defrauding merchant programs in the background. Overall this developer's extensions have nearly 70K users.

Another extension we found named "Facebook Theme: Basic Minimalist Black Theme" (2.5K users) allows users to change the appearance of Facebook. Besides its stated intent, however, it also monitors browsing and appends an affiliate identifier to 7 different Amazon sites. By using its Content Security Policy (Section 2.3) to perform `eval`, it runs a highly-obfuscated hexadecimal and base64-encoded background script that stores all affiliate identifiers in Chrome's storage (using `storage` permissions), and registers callbacks on tab update events using `tab` permissions. When the user visits any URL, Chrome notifies the extension, and the extension uses regular expressions to identify target Amazon URLs for which to add an affiliate identifier. The extension then updates the URL before the browser sends the request. The creator of the extension appears well aware that the extension violates Amazon's Conditions of Use [3] and has heavily used obfuscation, evidently to evade any static analysis for detecting affiliate fraud.

As another example, we found an extension named "Page Refresh" (200 installations) that allows users to refresh tabs periodically and only requests `tabs` permission. By using the background page to listen on all tab update events, if a user visits a merchant site it sets the URL in the tab to a URL shortener that redirects the user to the same merchant page but with the affiliate identifier included in the URL, thereby stuffing a cookie into the user's browser. This extension abuses 40 different merchants, again including Amazon.

This approach has the advantage that it capitalizes on organic traffic to merchant sites, which can make fraud detection difficult because merchants see visit behavior highly similar to that they would otherwise see as a result of legitimate affiliate referrals.

The second group of extensions includes extensions that clearly state in their descriptions that the extension monetizes the user's online purchases—generally for charitable causes or donations to organizations. The intent or legitimacy of such programs is difficult to ascertain. For example, the extension "Give as you Live" [8] has over 11K users, and forms part of a larger campaign [7] to raise funds for charities from user purchases online. The extension works by adding a list of stores for which the extension author has signed up as an affil-

iate to the results of major search engines. It also adds a script on merchant sites such as `amazon.co.uk` to redirect users via its own URL. While it does bring legitimate and likely well-intentioned traffic to Amazon, the legitimate affiliates can lose out if users choose to read product reviews on affiliate sites and then make the purchase via this extension.

In fact, a plethora of extensions exists allowing users to donate to charity simply by shopping online. Another such extension uses `webRequest` permissions to modify the requested URL to the affiliate URL, including over-writing the existing affiliate URL. While this clearly constitutes cookie-stuffing, the extension advertises itself as "Help support our charity by shopping at amazon.co.uk".[3]

5.3 Information Theft

Information theft clearly reflects malicious behavior that has the potential to harm the user in a number of ways, from disclosing private information to financial loss. This broad category of abuse in many ways replicates the functionality of some malware families. Within the browser, we observe stealing of: keypresses, passwords and form data, private in-page content (e.g., bank balances), and authentication tokens such as cookies. We do not include extensions that simply re-use existing authentication tokens already present, such as extensions that spam on social networks; we discuss these in Section 5.4.

One example of keylogging we found in the Chrome Web Store, "Chrome Keylogger", is an experimental extension from researchers [14] that is now removed. Keyloggers use content scripts to register callbacks for key press events, recording the pressed key by using the messaging API to communicate with a background page. The background page then queues up data to send to a remote server. This behavior has similarities with that of extensions that steal form data, although the specific event handlers differ. Both form field theft and keylogging require the extension to specify a content script but do not require other permissions.

5.4 OSN Abuse

Online social network abuse constitutes the final category of prevalent malicious extensions we found. These extensions typically target Facebook, and spread via both the Chrome Web Store and sideloading. These extensions use existing authentication data to interact with the APIs and websites of online social networks. Previous work identified and reported Chrome extensions that abuse social networks, reporting that thousands of users had installed extensions from the Chrome Web Store that spam on Facebook [4].

[3] The extension creator also helpfully marked the JavaScript code that adds the affiliate identifier as something to obfuscate in the future.

```
"content_scripts": [{
  "js":["BlobBuilder.js", ... ],
  "matches":["http://*/*", "https://*/*" ],
  "run_at": "document_end"
}],
"permissions":["http://*/*", "https://*/*",
"*://*.facebook.com/",
"tabs", "cookies", "notifications",
"contextMenus", "webRequest", ...],
```

Figure 3: Permissions and content script excerpts from the manifest for an extension that spams on Facebook and creates Tumblr accounts.

We found a number of extensions that post spam messages and use other features provided by social networks, such as the ability to upload and comment on photos or query the social graph. When we execute these extensions with Hulk, the HoneyPage features allows the extensions to create elements and insert them into the DOM tree. While we do not typically inspect the visual results of our executions, in one case we observed an extension creating `div` elements to mimic Facebook status updates and inserting them into a page. The HoneyPage acted as a sink for the spam status messages resulting in a page full of spam for the infected user.

One extension of interest, *WhasApp* (a name closely resembling the popular *WhatsApp*, a mobile chat application), has since been removed from the Chrome Web Store, but we also found evidence of the same extension being sideloaded from malware. The extension targets both Facebook and Tumblr. At Facebook, the extension uploads images to Facebook and then comments on them with messages containing URLs. In some cases the links are used to spread the malicious extension to a wider audience, while other URLs sought to monetize users as part of a spam campaign to advertise products. At Tumblr, the extension creates new Tumblr accounts and verifies them in the background.

The manifest file contains permissions and content scripts that request broad access, as shown in Figure 3. The extension is in fact over-privileged, since the extension in fact does not use some of the API permissions the manifest includes. Prior work has identified over-privileging as not uncommon, even among benign extensions [13]. Figure 3 shows the extension specifically requesting access for permissions and content scripts on `facebook.com` in addition to all other sites, which provides a hint as to the sites targeted. To carry out spamming on Facebook and Tumblr account creation,

the extension actually only requires the use of content scripts. The abusive component of the extension is 15 lines of JavaScript that downloads a much larger remote JavaScript file containing the spamming functionality.

6 Recommendations

In this section, we frame changes to make Chrome's extension ecosystem safer. Extensions should not have the ability to manipulate browser configuration pages, such as `chrome://extensions`, that govern how users manage and uninstall extensions. Extensions should also not be allowed to uninstall other extensions unless they are from the same author or a trusted source (such as Google or Antivirus vendors). We also recommend preventing extensions from manipulating HTTP requests by **removing security-related headers** that compromise the security of web pages. This change will require modifications to several extension APIs to comprehensively address this issue, the primary one being `webRequest`.

To address cloaking and other changes in remotely included content, we suggest that Google should encourage **local inclusion of static files** in the context of a web page. Chrome supports pushing automatic updates of extensions to users, so remotely including additional JavaScript code is not necessary to support rapid changes in an extenion's code. This change will make it possible to have a more complete analysis of extension behavior, since the analysis engine—Hulk or otherwise[4]—will have the complete extension code available. To encourage developers to write completely self-contained extensions and not load additional code from the network, one could introduce a new policies, such as: if an extension loads code from a remote site, it loses permissions such as the ability to inject that new code into the visited pages.

Finally, extensions should not have the ability to **hook all keyboard events** on a given site. The *window.onkey** API that exists in JavaScript has utility for pages that want to intercept the keyboard events of their users, but in the context of extensions it provides too much power. An experimental API (*chrome.commands*) exists that allows extensions to register keyboard shortcuts; this strikes us as a step in the right direction, as this covers the common use-case for requiring access to these events.

These suggestions will not eliminate malicious extensions, but can prevent classes of attacks, and significantly facilitate the analysis of extensions.

[4] In particular, ultimately an extension store operator such as Google needs to undertake such analysis as part of its curation of the store contents.

7 Limitations

Our system uses dynamic analysis for analyzing extensions, and, as with every dynamic analysis system, the correct classification of an extension relies on triggering the malicious activity. Hulk employs HoneyPages and event handler fuzzing on the extension's web request listeners to enhance dynamic analysis, but does not provide a complete view of extension behavior. For example, we do not attempt to address cloaking that loads different code based on the client's location or time. We also will not observe behavior that depends on specific targets, such as those that require user interaction with a visited page to take effect. Similarly, pages that require sign-in pose difficulties. Hulk has a pre-set list of sites and credentials to use while visiting pages, but does not perform account creation on the fly.

Hulk's Honeypages do not currently support multi-step querying of DOM elements. While we can place elements in the DOM tree that an extension looks for, if the extension expects elements to have additional properties in order to trigger its malicious behavior, we will fail to adapt to the extension's expectations. We plan on improving HoneyPages to support multi-step querying, and for many element types and attributes this appears possible.

We currently also lack data flow analysis in the Chrome browser, a feature that would substantially improve the depth of behavior available for analysis. One example where this would prove particular useful regards keystroke interception. Without data flow tracking, we cannot automatically derive whether this information ultimately becomes transmitted to a third party via a network request.

Another difficult concern for Hulk is analysis evasion by extensions that specifically look for HoneyPages. A determined adversary with knowledge of the system could try to evade Hulk by querying for random elements in the DOM tree first, and, if found, avoid malicious activity. A similar type of evasive behavior arose for in submissions to Wepawet [17]. One way to counter this is by introducing non-deterministic HoneyPages for which DOM tree queries only succeed with a given probability. We could further enhance this approach by crawling a few million sites and building models of the existing elements to assign apt probabilities weights for different queries. This approach may also require analysis of an extension's DOM queries in case the extension repeatedly performs these in an effort to detect randomized queries. Finally, we can consider measuring code coverage to examine the impact that each DOM query has on the amount of code executed by an extension, as the extension will skip executing the malicious code when it detects the presence of an analysis system.

8 Related Work

Browser extensions have been available for Internet Explorer and Firefox for over a decade. As a result of a study of vulnerabilities in Firefox extensions, Barth et al. designed an extension architecture that promotes least privilege and isolation of components to prevent a compromised extension from gaining full access to a user's browser [5], an architecture subsequently adopted by Google Chrome. Since then, further work has examined the success of the Chrome extension architecture at preventing damage [6] and the ability of developers to correctly request privileges for their extensions [13]. Similar studies have examined the Firefox extension system to limit the potential damage arising from exploitation of extension vulnerabilities, and to improve the defenses the browser provides [27]. These works have a focus mostly tangential to our work, since the principle of least privilege does not prevent an overtly malicious extension from executing malicious code.

The security industry has documented malicious extensions in ways similar to malware reports and other new threats [2, 4]. Liu et al. examined Google Chrome extensions and, based on malicious extensions the authors built, suggested refined privileges to make detecting malicious extensions easier [21]. In our work, we build a system that performs dynamic analysis and classification of extensions, and present an analysis of malicious extensions that we found in the wild.

JavaScript-based program analysis has particular promise for benefiting our work, and in light of our current limitations we will be exploring techniques that we can adapt to improve our system's detection capabilities. Research has applied information flow analysis to Firefox extensions [10], performed taint-based tracking of untrusted data within the browser [11], used symbolic execution to detect vulnerabilities [26], applied static verification to extensions [16], contained extensions in privacy-preserving environments [20], and used supervised learning of browser memory profiles to detect privacy-sensitive events [14].

Our work has similarities to that of other malware detection and execution systems. While our implementation and requirements significantly differ from systems that execute Windows binary malware (such as Anubis [1]), at a high level we share common goals of executing and extracting data from samples. Like Anubis, Wepawet, the GQ honeyfarm, and other malware execution platforms, we share the difficult problem of triggering malicious behavior in a synthetic environment [9, 19]. Other research in this area has focused on classification and discerning malware from goodware [24].

9 Summary

In this paper we presented Hulk, a system to dynamically analyze Chrome browser extensions and identify malicious behavior. Our system monitors an extension's actions and creates a dynamic environment that adapts to an extension's needs in order to trigger the intended behavior of extensions, classifying the extension as malicious or benign accordingly. In total, we identified 130 malicious and 4,712 suspicious extensions that have up to 5.5 million browser installations, many of which remain live in the Chrome Web Store. Based on these results, we developed a detailed characterization of the malicious behavior that we found, targeted at determining the motivation behind the extension. Finally, we propose several changes for the Chrome browser ecosystem that could eliminate classes of extension-based attacks and aid with analysis.

Acknowledgments

We would like to thank our shepherd David Evans for his insightful comments and feedback. We would also like to thank Niels Provos, Adrienne Porter Felt, Nav Jagpal and the rest of the Safebrowsing team at Google for their insight and discussions throughout this project. This work was supported by the National Science Foundation under grants 0831535 and 1237265, by the Office of Naval Research (ONR) under grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, by Secure Business Austria and by generous gifts from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Anubis — Malware Analysis for Unknown Binaries. http://anubis.iseclab.org/.

[2] AMADEO, R. Adware vendors buy Chrome Extensions to send ad- and malware-filled updates. http://arstechnica.com/security/2014/01/malware-vendors-buy-chrome-extensions-to-send-adware-filled-updates/, Jan 2014.

[3] AMAZON. Associates Program Operating Agreement. https://affiliate-program.amazon.com/gp/associates/agreement/, 2012.

[4] ASSOLINI, F. Think twice before installing Chrome extensions. http://www.securelist.com/en/blog/208193414/Think_twice_before_installing_Chrome_extensions, Mar 2012.

[5] BARTH, A., FELT, A. P., SAXENA, P., AND BOODMAN, A. Protecting Browsers from Extension Vulnerabilities.

In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2010).

[6] CARLINI, N., FELT, A. P., AND WAGNER, D. An Evaluation of the Google Chrome Extension Security Architecture. In *Proceedings of the USENIX Security Symposium* (2012).

[7] CHARLES ARTHUR. Infographic: Internet shopping. `http://www.theguardian.com/technology/blog/2011/jul/04/internet-shopping-infographic-give-as-you-live-charity`, 2011.

[8] CHROME WEB STORE. Give as you Live. `https://chrome.google.com/webstore/detail/give-as-you-live/fceblikkhnkbdimejiaapjnijnfegnii`, 2013.

[9] COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the World Wide Web Conference (WWW)* (2010).

[10] DHAWAN, M., AND GANAPATHY, V. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009).

[11] DJERIC, V., AND GOEL, A. Securing script-based extensibility in web browsers. In *Proceedings of the USENIX Security Symposium* (2010).

[12] F-SECURE. Coremex innovates search engine hijacking. `http://www.f-secure.com/weblog/archives/00002689.html`, April 2014.

[13] FELT, A. P., GREENWOOD, K., AND WAGNER, D. The Effectiveness of Application Permissions. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)* (2011).

[14] GIUFFRIDA, C., ORTOLANI, S., AND CRISPO, B. Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2012), ACM.

[15] GOOGLE. What are extensions? `https://developer.chrome.com/extensions/index`, 2014.

[16] GUHA, A., FREDRIKSON, M., LIVSHITS, B., AND SWAMY, N. Verified Security for Browser Extensions. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 115–130.

[17] KAPRAVELOS, A., SHOSHITAISHVILI, Y., COVA, M., KRUEGEL, C., AND VIGNA, G. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the USENIX Security Symposium* (2013).

[18] KAY, E. Protecting Chrome users from malicious extensions. `http://chrome.blogspot.com/2014/05/protecting-chrome-users-from-malicious.html`, May 2014.

[19] KREIBICH, C., WEAVER, N., KANICH, C., CUI, W., AND PAXSON, V. GQ: Practical containment for measuring modern malware systems. In *Proceedings of the ACM Internet Measurement Conference (IMC)* (2011), ACM, pp. 397–412.

[20] LI, Z., WANG, X., AND CHOI, J. Y. Spyshield: Preserving privacy from spy add-ons. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)* (2007).

[21] LIU, L., ZHANG, X., YAN, G., AND CHEN, S. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2012).

[22] LUDWIG, P. No more silent extension installs. `http://blog.chromium.org/2012/12/no-more-silent-extension-installs.html`, Dec 2012.

[23] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).

[24] RAJAB, M. A., BALLARD, L., LUTZ, N., MAVROMMATIS, P., AND PROVOS, N. CAMP: Content-Agnostic Malware Protection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2013).

[25] REDDIT. Reddit: I am One of the Developers of a Popular Chrome Extension.... `http://www.reddit.com/r/IAmA/comments/1vjj51/i_am_one_of_the_developers_of_a_popular_chrome/`, Jan 2014.

[26] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[27] TER LOUW, M., LIM, J. S., AND VENKATAKRISHNAN, V. Enhancing Web Browser Security Against Malware Extensions. *Journal in Computer Virology 4*, 3 (2008), 179–195.

[28] WANG, D., SAVAGE, S., AND VOELKER, G. M. Cloak and Dagger: Dynamics of Web Search Cloaking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011), ACM, pp. 477–490.

[29] WEST, M. An Introduction to Content Security Policy. `http://www.html5rocks.com/en/tutorials/security/content-security-policy/`, 2012.

Precise client-side protection against DOM-based Cross-Site Scripting

Ben Stock
FAU Erlangen-Nuremberg
ben.stock@cs.fau.de

Sebastian Lekies
SAP AG
sebastian.lekies@sap.com

Tobias Mueller
SAP AG
tobias.mueller07@sap.com

Patrick Spiegel
SAP AG
patrick.spiegel@sap.com

Martin Johns
SAP AG
martin.johns@sap.com

Abstract

The current generation of client-side Cross-Site Scripting filters rely on string comparison to detect request values that are reflected in the corresponding response's HTML. This coarse approximation of occurring data flows is incapable of reliably stopping attacks which leverage non-trivial injection contexts. To demonstrate this, we conduct a thorough analysis of the current state-of-the-art in browser-based XSS filtering and uncover a set of conceptual shortcomings, that allow efficient creation of filter evasions, especially in the case of DOM-based XSS. To validate our findings, we report on practical experiments using a set of 1,602 real-world vulnerabilities, achieving a rate of 73% successful filter bypasses. Motivated by our findings, we propose an alternative filter design for DOM-based XSS, that utilizes runtime taint tracking and taint-aware parsers to stop the parsing of attacker-controlled syntactic content. To examine the efficiency and feasibility of our approach, we present a practical implementation based on the open source browser Chromium. Our proposed approach has a low false positive rate and robustly protects against DOM-based XSS exploits.

1 Introduction

Ever since its initial discovery in the year 2000 [6], Cross-Site Scripting (XSS) is an ever-present security concern in Web applications. Even today, more than ten years after the first advisory, XSS vulnerabilities occur in high numbers [39] with no signs that the problem will be fundamentally resolved in the near future. Furthermore, in recent years, DOM-based XSS, a subtype of the vulnerability class that occurs due to insecure client-side JavaScript, has gained traction, probably due to the shift towards rich, JavaScript heavy Web applications. In a recent study, we have shown that approximately 10% of the Alexa Top 5000 carry at least one DOM-based XSS vulnerability [18].

The design of protection measures against XSS has received considerable attention. In its core, XSS is a client-side security problem: The malicious code is executed in the client-side context of the victim, affecting his client-side execution environment. Hence, a well-suited place to protect end users against XSS vulnerabilities is the Web browser. Following this concept, several client-side XSS filters have been developed over the years.

These contemporary client-side XSS filtering mechanisms rely on string-based comparison of outgoing HTTP requests and incoming HTTP responses to detect reflected XSS attack payloads. In essence, this string comparison is an approximation of server-side data flows that might result in direct inclusion of request data in the HTTP response. While this approximative approach is valid for server-based XSS vulnerabilities – the browser has no insight on the server-side logic – it is unnecessarily imprecise for client-side XSS issues.

In contrast to server-side problems, the complete data flow within the browser from attacker-controlled sources to the security-sensitive sinks into the browser's JavaScript engine occurs within one system and thus can be tracked seamlessly and precisely. Based on this observation, we propose a different protection approach: A client-side XSS filtering mechanism relying on precise dynamic taint tracking and taint-aware parsers within the browser.

To demonstrate the current limitations of the established approaches – which focus mainly on stopping reflected XSS attacks – we first conduct an in-depth analysis of the current state-of-the-art in client-side XSS filtering, with focus on the capabilities of thwarting DOM-based XSS attacks (see Section 3). In course of this analysis, we uncover a set of conceptual weaknesses which, taken together, render the existing techniques incapable of protecting against the majority of client-side XSS attacks (see Section 4). To practically validate our analysis, we report on a fully automatic system to create XSS attacks which evade the current protection mechanism:

Using a data set of 1,602 real-life DOM-based XSS vulnerabilities, we successfully created XSS vectors that bypassed client-side filtering in 73% of all cases, affecting 81% of all vulnerable domains we found.

Motivated by the results of our experiments, we propose an alternative approach for client-side prevention of DOM-based XSS: Using character-level taint tracking in the browser we can precisely detect cases in which attacker-controlled values end up in a syntactic parsing context which might lead to the browser executing the injected data as JavaScript. By enhancing the browser's HTML and JavaScript parsers to identify tokens that carry taint markers, we can efficiently and robustly stop injection attempts. To summarize, we make the following contributions:

- *Systematic analysis of conceptual shortcomings of current client-side XSS filters:* We report on a systematic investigation on the current state-of-the-art client-side XSS filter, the XSS Auditor, and identify a set of conceptual flaws that render the filter incapable of effectively protecting against DOM-based XSS attacks.
- *Automatic filter bypass generation:* To validate our findings and to demonstrate the severity of the identified filter limitations, we built a fully automated system to generate XSS payloads which bypass the string comparison based XSS filter. Our system leverages the precise flow information of our taint-enhanced JavaScript engine and our detailed knowledge on the Auditor's functionality to create exploit payloads that are tailored to a vulnerability's specific injection context and the applicable filter weakness. By practically applying our system to a set of 1,602 real-world DOM-based XSS vulnerabilities, we achieved a success rate of 73% successful filter bypasses.
- *Robust protection approach, utilizing client-side taint propagation:* Based on the identified weaknesses in the established XSS filtering approaches, we propose an alternative protection measure which is designed to address the specific characteristics of DOM-based XSS. Through combining a taint-enhanced browsing engine with taint-aware JavaScript and HTML parsers, we are able to precisely track the flow of attacker-controlled data into the parsers. This in turn enables our system to reliably detect and stop injected code on parse time.

2 Technical Background

In the following, we briefly discuss DOM-based Cross-Site Scripting and shed light on the technical basis used for this work, namely a taint-aware browsing engine.

2.1 DOM-based Cross-Site Scripting

Cross-Site Scripting (XSS) is a term describing attacks where the adversary is able to inject his own script code into a vulnerable application, which is subsequently executed in the browser of the victim in the context of this application. In contrast to the server-side variants of XSS, namely reflected and persistent, the term DOM-based Cross-Site Scripting (or DOM-based XSS) subsumes all classes of vulnerabilities which are caused by insecure client-side code. The term itself was coined by Klein in 2005 [16]. These issues come to light when untrusted data is used in a security-critical context, such as a call to `eval`. In the context of DOM-based XSS, this data might originate from different sources such as the URL, postMessages [38] or the Web Storage API.

2.2 Browser-level Taint Tracking

One of the underlying technical cornerstones of this paper is the taint-enhanced browsing browsing engine we developed for CCS 2013 [18]. This engine allows precise tracking of data flows from attacker-controlled sources, such as `document.location`, to sinks, such as `eval`.

Our implementation, based on the open source browser Chromium, provides support for tracking information flow on the granularity of single characters by attaching a numerical value to identify the origin of the character's taint. This taint marker is propagated whenever string operations are conducted and is also persisted between the two realms of the rendering component, i.e., Blink, and the V8 JavaScript engine.

As this part of our system is not one of the paper's major contributions, we omit further details for brevity reasons and refer the reader to the aforementioned paper.

3 Current Approaches for Client-side XSS Filtering

In this section we investigate the current in-browser techniques used to detect and prevent XSS attacks. More specifically, we describe the concepts of the Firefox plugin NoScript [20], Internet Explorer's XSS Filter [29] and Chrome's XSS Auditor [2].

3.1 Regular-expression-based Approaches: NoScript and Internet Explorer

One of the first mechanisms on the client side to protect against XSS attacks was introduced by the NoScript Firefox Plugin [22] in 2007. NoScript utilizes regular expressions to filter outgoing HTTP *requests* for potentially malicious payloads. If one of the regular expressions matches, the corresponding parts are removed from the HTTP request. The malicious payload will thus never

reach the vulnerable application and hence an attack is thwarted. Nevertheless, as described in NoScript's feature list, this potentially leads to false positives [21] due to its aggressive filtering approach. NoScript works around this issue by prompting the user whether to repeat the request, this time disabling the protection mechanism. While this seems to be a valid approach for NoScript's security-aware users, it is not acceptable as a general Web browser feature, as many studies have shown that an average user is not able to properly react to such security warnings [9, 13, 34].

In order to tackle this problem Microsoft slightly extended NoScript's approach and integrated it into Internet Explorer [29]. Similar to NoScript, IE's XSS filter utilizes regular expressions to identify malicious payloads within outgoing HTTP requests. Instead of removing the potentially malicious parts from a request, the filter generates a signature of the match and waits for the HTTP response to arrive at the browser. If the signature matches anything inside the response, i.e., if the payload is also contained within the response, the filter removes the parts it considers to be suspicious. Thus, attacks are only blocked if the payload is indeed contained in the response and, hence, depending on the situation, false positives are less frequent. In fact, avoiding false positives is one of the filter's many design goals [30], even if this results in a higher false negative rate, as Microsoft's David Ross states: "Thus, the XSS Filter defends against the most common XSS attacks but it is not, and will never be, an XSS panacea." [30].

In 2010, Bates et al. [2] demonstrated that regular-expression-based filtering systems have severe issues and proposed a superior approach in the form of the XSS Auditor, which has been adopted by the WebKit browser family (Chrome, Safari, Yandex).

3.2 State-of-the-Art: The XSS Auditor

Based on the identified weaknesses of regular-expression-based XSS defenses, Bates et al. proposed the XSS Auditor – a new system that is "faster, protects against more vulnerabilities, and is harder for attackers to abuse" [2]. Up to now, the XSS Auditor constitutes the state-of-the art in client-side XSS mitigation, albeit focusing mainly on reflected XSS.

As we will demonstrate in this paper, the XSS Auditor also has shortcomings, especially related to DOM-based XSS attacks. Before we explore the limitations of the system in the next section, we provide an overview of the Auditor's protection mechanism.

One of the key differences between Chrome's XSS Auditor and previous filter designs is the filter's placement within the browser architecture. Instead of applying regular expressions on the string representations of the HTTP requests or responses, the Auditor is placed between the HTML parser and the JavaScript engine [2]. The idea behind this placement is, that an attacker's payload has to be parsed by the HTML parser to be transferred to the JavaScript engine where the injected payload is being executed.

In order to block XSS attacks, the Auditor receives each token generated by the HTML parser and checks whether the token itself or some of its attributes are contained in either the request URL or the request body. If so, the filter considers the token to be injected and replaces JavaScript or potentially harmful HTML attributes with a benign value. Such a benign value is a payload that has no effect, such as `about:blank`, `javascript:void(0)` or an empty string. The injected fragments will thus not be passed to the JavaScript engine and hence attacks are prevented.

The main design goals of the filter are to avoid false positives and to minimize performance impact. Before demonstrating that these goals severely impact the filter's detection capabilities, we will first provide details on the detection algorithm (simplified to satisfy space and readability constraints):

1. **Initialization (For document fragments)**
   (a) Deactivate the filter
2. **Initialization (For each full document)**
   (a) Fully decode the request URL
   (b) Fully decode the request body
   (c) Check if request could contain an injection
      i. If not, deactivate the filter
      ii. Otherwise continue
3. **For each start token in the document do...**
   (a) Check and delete dangerous attributes
      i. Delete injected event handlers
      ii. Delete injected JavaScript URLs
   (b) Conduct tag specific checks
4. **For each script token in the document do...**
   (a) Check and delete injected inline code

As soon as the so-called *HTMLDocumentParser* is spawned by Chrome, an initialization routine of the XSS Auditor is called. The parser can either be invoked for parsing document fragments or complete documents. While the XSS filter is deactivated for document fragments, it guesses whether an injection attack is likely to be present for full documents. If either the URL or the request body contains one of the characters shown in Listing 1, the filter is activated. If none of these characters is found, the filter assumes the browser not being under attack and skips the complete filtering process.

If, on the other hand, one of the characters mentioned in Listing 1 is present in the request the Auditor investigates every token within the document for injected values that might cause script execution. This process

Listing 1 Required characters to activate the filter

```
static bool isRequiredForInjection(UChar c)
{
    return (c == '\'' || c == '"' ||
            c == '<'  || c == '>');
}
```

is threefold: First the Auditor looks for dangerous attributes, second it conducts tag specific checks for certain attributes and third it filters injected inline scripts.

Dangerous Attributes are, in the view of the Auditor, attributes that either contain a JavaScript URL or have the name of an inline event handler (`onclick`, `onload`, etc.) as these attributes can enable XSS attacks. If such an attribute is found, the Auditor searches for it within the corresponding request. If a match is found, the filter assumes the attribute to be injected and either deletes the complete attribute value in case of event handlers or replaces the JavaScript URL with a benign URL.

Tag-specific filtering Besides event handlers and attributes containing JavaScript URLs, other tag specific attributes that need to be filtered exist. An attacker could, for example, inject a script tag and use the `src` attribute to load an external script file. Hence, for any script token, the Auditor additionally checks the legitimacy of the `src` attribute. In total, the Auditor conducts such checks for 18 additional attributes contained in 11 tokens (script, object, param, embed, applet, iframe, meta, base, form, input and button).

Filter inline scripts Whenever the Auditor encounters a script tag, it also validates whether the content between opening and closing tag has been injected. If the content can be found in the request, it is replaced with an empty string.

4 Limitations of String-based XSS Filters

In this section we report on a detailed analysis we conducted to assess the XSS Auditor's protection capabilities with a focus on DOM-based XSS. Although the XSS Auditor was designed to stop reflected Cross-Site Scripting attacks, it is also the most advanced and deployed filter against DOM-based XSS attacks. In the following, we therefore analyze issues related to the concept of the Auditor, which impair its capabilities of stopping DOM-based XSS attacks. In doing so, we show the arising need for a filter capable of stopping DOM-based XSS attacks.

4.1 Scope-related Issues

In general, the Auditor is called whenever potentially dangerous elements are encountered during the initial parsing of the HTTP response. These are, however, not the only situations in which attacker-controlled data might end up being interpreted as code. In this section, we explore situations in which the filter is not active and hence does not protect against attacks.

eval As mentioned earlier, the Auditor is placed between the HTML parser and the JavaScript engine to intercept potential XSS payloads. Still, not every DOM-based XSS attack needs to go through the HTML parser. If a Web site invokes the JavaScript function `eval` with user-provided data, the execution will never pass the HTML parser. Therefore, the Auditor will never see a malicious payload that an attacker injected into a call to `eval`. As we will demonstrate later, `eval` is commonly used in Web applications.

innerHTML While script tags inserted via `innerHTML` are not executed, it is still possible to execute JavaScript via inline event handlers. Hence, `innerHTML` is also prone to XSS attacks. In earlier versions of the Auditor content parsed via `innerHTML` was also filtered. Google later experienced some performance drawbacks in innerHTML-heavy applications [17] and as a consequence, the Auditor is nowadays disabled for document fragment parsing which is invoked upon an assignment to `innerHTML`.

Direct assignment to dangerous DOM properties Besides `eval` and `innerHTML` it is also possible to trigger the execution of scripts without invoking a HTML parsing process as a few examples in Listing 2 show. As no HTML parsing takes place in these cases, the XSS Auditor is never invoked. Hence, if a Web application assigns a user-controlled value to such a DOM property, an attacker is able to evade the filter.

Listing 2 Examples for dangerous DOM properties

```
var s = document.createElement("script");
s.innerText = "myFunction(1)";        // 1.
s.src = "http://example.org/script.js"// 2.

var i = document.createElement("iframe");
i.src = "javascript:myFunction(1)"    // 3.

var a = document.createElement("a");
a.href = "javascript:myFunction(1)"   // 4.
```

Second order flows When investigating a token, the Auditor always validates whether a suspicious value was contained within the preceding HTTP request's URL or body. As demonstrated by Hanna et al. [10], second order flows are relevant for DOM-based XSS. So, for example, if a value is written into LocalStorage within one request/response cycle, it can be used to cause a DOM-based XSS attack in another request/response pair. As the Auditor only investigates the last request, it will not find the value sent with the second-last request. LocalStorage is only one of many ways to persist data across multiple HTTP requests as Cookies, WebStorage or the File API exist nowadays.

Alternative attack vectors It is not sufficient to only check the URL and the request body in order to prevent DOM-based XSS attacks. Multiple other sources of attacker-controllable data exist which could be abused to inject malicious content into an application. Examples are the PostMessage API, the `window.name` attribute, or the `document.referer` attribute. As the Auditor does not take these sources into account, they can be used to evade the filter.

Furthermore, Bojinov et al. demonstrated that data can be injected by an attacker via alternative communication channels [4]. Thus, so-called cross-channel scripting attacks also bypass the XSS Auditor.

Unquoted attribute injection During initialization, the Auditor checks whether filtering is necessary by verifying the presence of the characters shown in Listing 1. In doing so, it implicitly assumes that an attack is not possible without these characters. This assumption, however, is wrong. In Listing 3 we show a common vulnerability and the corresponding attack (note: the value of the id attribute is not surrounded by quotes). In this example, the payload does not make use of the required characters. Normally, the XSS Auditor would block the `src` attribute containing the JavaScript URL. In this case, however, it does not conduct any checks as it is deactivated.

Listing 3 Unquoted Attribute injection

```
var id = location.hash.slice(1);
var code = "<iframe id=" + id + " [...]>";
    code += "</iframe>";
document.write(code);

// attack payload within URL
"//example.org/#1 src=javascript:eval(name)"
```

4.2 String-matching-related Issues

In the following we explore the limits of the implemented string matching algorithms. Whenever the Auditor finds a potentially dangerous element or attribute, it verifies whether the corresponding string representation can be found in the request. If an attacker is able to mislead the string-matching algorithm, the filter can be bypassed. Hence, the precision of this process determines the filter's effectiveness and as a result its false positive and false negative rates.

4.2.1 Partial Injections

One of the assumptions the Auditor makes is that an attacker has to inject a complete tag or attribute to successfully launch an attack. As a consequence the filter always aims to find the complete tag or the complete attribute within the request. While this approach reduces false positives as it is very unlikely that an application contains an existing tag or attribute in its URL legitimately, it does not regard application-specific scenarios. This assumption leads to potential problems in three different cases:

Attribute Hijacking One of the first things the Auditor does is to check whether a dangerous attribute was injected into the application. Hence, whenever it discovers a dangerous attribute during the parsing process it regenerates the string representation of the attribute and matches it against the URL and the request body. Listing 4 shows the string generation process:

Listing 4 Attribute string matching

```
// current start token
<iframe [...] onload="alert('example')">
// Step 1: extract the dangerous attribute
onload="alert('example')"
// Step 2: Truncate after 100 characters
onload="alert('example')"
// Step 3: Truncate at a terminating char
onload="alert('
```

After detecting a potentially dangerous attribute the Auditor extracts its decoded string representation. Then, it truncates the attribute at 100 chars to avoid the comparison of very long strings. It finally truncates the string at one of seven so-called terminating characters (this is done to detect attacks, that we will cover later). The resulting string is then matched against the URL. Obviously, the resulting string always contains the name of the potentially dangerous attribute. Hence, the underlying assumption here is that the attacker always has to inject the attributes herself. In real-world applications, however, attributes can often be hijacked by an attacker as shown in Listing 5. Although the `onload` attribute is a dangerous event handler attribute, the Auditor will not discover it within the URL as the `onload` attribute's name is hardcoded within the application and not injected by the attacker.

Listing 5 Attribute & Tag hijacking vulnerability

```
var h = location.hash.slice(1);
var code = "<iframe onload='" + h + "'"
    code += "[...]></iframe>";
document.write(code);

//attack for attribute hijacking
"//example.org/#alert('example')"
//attack for tag hijacking
"//example.org/#' srcdoc='...'"
```

Tag Hijacking After checking for dangerous attributes the Auditor conducts tag specific attribute checks. Matching all attributes of all tokens within an HTML

document against the URL and request body, however, can be a very time consuming and error-prone task. Therefore, the auditor only matches an attribute against the URL if it can find the tag's name in the URL. For example, if the filter investigates an iframe token it validates whether the sequence <iframe is contained in the request before matching the src or srcdoc attribute [1]. Hence, if the injection point of a vulnerability lies within such a tag, the attacker can hijack the tag and inject additional attributes to it. As the tag itself is hardcoded the Auditor will skip any of its checks for specific attributes. An example of this attack is provided in Listing 5.

In-Script Injections Another vulnerability that is not detectable by the XSS Auditor is an injection inside of an existing inline script. As described in Section 3.2, whenever the filter encounters a script tag, it matches the *complete* inline content of the script against the request. Real-world Web applications however often make use of dynamically generated inline scripts made up from user-controllable input mixed with hardcoded values. Hence, instead of injecting a script tag via the URL an attacker is able to simply inject code into an existing dynamic inline script. As a consequence searching for the complete script content within sources of user input will not be successful.

4.2.2 Trailing Content

A very similar problem to partial injections is trailing content. When real-world Web applications write input to the document, they do not simply write one single value coming from the user but rather use a string that was constructed from hardcoded values as well as potentially attacker-controlled values. Listing 6 shows a real-world example.

Listing 6 An example of String construction

```
var code = "<iframe src='//example.org/";
    code += getParamFromURL("page_name");
    code += ".html'></iframe>";;
document.write(code);

// attack payload:
"' onload='alert(1);foo"
// resulting code
"<iframe src='//example.org/'
       onload='alert(1);foo.html'>"
```

Note, that the injection point is inside the src attribute of the iframe tag. Within this src attribute, the attacker-controllable input starts in the middle of the attribute (after //example.org/) and some more content is following the injection point (.html). When crafting an attack, the attacker is able to use the trailing content within the payload to confuse the string matching process. Despite the fact that the Auditor is aware of this issue (source code comments indicate this) and defends against it, the current defenses are not able to reliably detect which parts are actually injected by the attacker and which parts are hardcoded within the Web application. We found four bypasses which allow an attacker to exploit this problem in different and partly unexpected ways. Due to the high complexity and the limited space, we omit a detailed explanation here.

4.2.3 Double Injections

Another conceptional flaw of string-matching-based approaches is the inability to discover concatenated values coming from more than one source of user input. As we have shown in previous work [18], a call to a security sensitive function contains on average three potentially attacker provided substrings. Listing 7 shows an example for such a double injection.

Listing 7 An example of double injection

```
var id = getParamFromURL("id");
var name = getParamFromURL("name");
var code = "<iframe id='" + id + "'";
    code += " name='" + name +"'";
    code += "[...]></iframe>";
document.write(code);

// attack
id="'/><script>void('"
name="');alert(1)</script>"
// resulting code
<iframe id=''/>
<script>void(' name=');alert(1)</script>
[...]></iframe>
```

As the call to document.write contains two injection points (id, name) an attacker is able to split the payload. A specially crafted set of inputs, as shown in the Listing, therefore leads to the creation of a valid script tag that is a combination of both attacker inputs. In this case, the Auditor's string matching algorithm would search for void('name=');alert(1) within the request. Finding this value in the URL, however, is not possible as the ' name =' part is hardcoded and not originating from the URL. Furthermore, the attacker is able to arbitrarily change the order in which the values appear within the URL. Hence, double injections are a severe conceptional problem for string-matching-based approaches. In total, we identified three different classes of double injection. The first class has been explained in the example above. A call to document.write contains two injection points

[1] For iframe.srcdoc the tag hijacking attack is not possible anymore, as concurrent research discovered this issue and reported it to Google. Upon the report Google changed the behavior for srcdoc. Nevertheless, for any other of the 18 special attributes, tag hijacking still is an issue

and the injected values are independent from each other. Very similar to this approach, the double injection pattern also applies to situations in which a single value is used twice within a single call to a security sensitive function. Finally, double injection attacks can be conducted if subsequent calls to `document.write` are made containing attacker-controllable values.

4.2.4 Application-specific Input Mutation

Another assumption of the XSS Auditor is that input of the user always reaches the parser without any modifications. If even one character of the input changed, the string matching algorithm will fail to find the payload and hence is not able to block the resulting attack. Application-specific encoding functions or data formats, therefore, lead to situations in which the filter can be bypassed.

4.3 Practical Experiments

As previously demonstrated we found numerous conditions under which the protection mechanisms of the XSS Auditor can be evaded, especially with respect to DOM-based Cross-Site Scripting. In order to assess the severity of the identified issues for real-world applications, we conducted a practical experiment. We used the methodology applied for our previous paper [18] to collect a set of 1,602 unique real-world DOM-based XSS vulnerabilities on 958 domains. We then built a bypass generation engine to verify whether a certain vulnerability allows employing one of the bypassing techniques described above.

Using our taint-aware infrastructure we are able to determine the exact injection context of a vulnerability. As soon as our infrastructure detects a call to a security sensitive sink such as `document.write`, `eval`, or `innerHTML`, it stores the string value and the exact taint information. Using a set of patched HTML and JavaScript parsers, we can exactly determine the location of the injection point. Using this data, we cannot only give an indication for a filter evasion possibility, but also generate an exact bypass that takes the injection point's context as well as the specific flaws of the Auditor into account. Applying this technique we compiled a set of bypasses that we evaluated against the vulnerabilities.

In doing so, we were able to bypass the filter for 73% of the 1,602 vulnerabilities, successfully exploiting 81% of the 958 domains in our initial data set.

4.4 Analysis & Discussion

As demonstrated by our practical experiments, the XSS Auditor – which aims at stopping reflected Cross-Site Scripting – can not stop DOM-based XSS attacks in the aforementioned cases. We therefore believe that additional defenses are necessary to combat this type of Cross-Site Scripting. Furthermore, the results of our analysis lead us to believe that the design of the XSS Auditor is prone to being bypassed in certain reflected XSS attack scenarios which are related to string-based matching issues. Since the focus of our work is on DOM-based XSS, we leave the investigation of this assumption to future work.

In our analysis, we identified two conceptual issues that limit the Auditor's approach in detecting and stopping DOM-based XSS attacks.

Placement One of the Auditor's strengths compared to Internet Explorer's and NoScript's approach is its placement between the HTML parser and the JavaScript engine. This way the Auditor does not need to approximate the browser's behavior during the filtering process. As we have shown in Section 4.1 the current placement is prone to different attack scenarios which are not taken into account by the filter. Currently the Auditor is not able to catch JavaScript-based injection attacks and situations in which HTML parsing is not conducted prior to a script execution.

String matching Even if it would be possible to extend the Auditor's reach to the JavaScript engine and the so-called *DOM bindings*, the string matching algorithm is another conceptual problem that will be very difficult if not impossible to solve. In order to cope with the situation the XSS Auditor introduced many additional checks and optimizations to thwart attacks. Nevertheless and despite the fact that a lot of bug hunters regularly investigate the filter's inner workings, we were able to find 13 bypasses targeting the string matching algorithm. All the mentioned problems will not disappear as employing string matching is inherently imprecise.

5 Preventing Client-side Injection Attacks during Parse-time

As the previous section has shown, current concepts of Cross-Site Scripting filters are not designed to thwart DOM-based XSS and, thus, are not sufficient to protect users against these kinds of attacks. In this section, we discuss the methodology behind our newly proposed filter as well as the corresponding policy considerations. We then go into detail on the issue of handling postMessages in our filter and finally outline the technical challenges we had to overcome to implement the concept into a real-world browser.

5.1 Methodology Overview

As we have demonstrated in Section 4.2, client-side XSS filters relying on string comparison lack the required precision for robust attack mitigation. String comparison

as an approximation of occurring data flows is a necessary evil for flows that traverse the server. For DOM-based XSS, this is not the case: The full data flow occurs within the browser's engines and can thus be observed precisely. For this reason, we propose an alternative protection mechanism that relies on runtime tracking of data-flows and taint-aware parsers and makes use of two interconnected components:

- A taint-enhanced JavaScript engine that tracks the flow of attacker-controlled data.
- Taint-aware JavaScript and HTML parsers capable of detecting generation of code from tainted values.

This way our protection approach reliably spots attacker-controlled data during the parsing process and is able to stop cases in which tainted data alters the execution flow of a piece of JavaScript. In the following, we discuss the general concept and security policy, whereas we go into more detail on the implementation in Section 5.4 and investigate the implications of our proposed filtering approach in Section 6.1.

5.2 Precise Code Injection Prevention

As we outlined in the previous section our protection approach relies on precise byte-level taint tracking.

In the following we give a detailed overview on the necessary changes we performed in order to implement our filtering approach. More specifically, we made changes to the browser's rendering engine, the JavaScript engine and the DOM bindings, which connect the two engines.

JavaScript Engine When encountering a piece of JavaScript code, the JavaScript engine first tokenizes it to later execute it according to the ECMAScript language specification.

While it is a totally valid use case to utilize user-provided data within data values such as String, Boolean or Integer literals, we argue that such a value should never be turned into tokens that can alter a program's control flow such as a function call or a variable assignment. We therefore propose that the tokenization of potentially attacker-provided data should never result in the generation of tokens other than literals. As our JavaScript engine is taint-aware, the parser is always able to determine the origin of a character or a token. Hence, whenever the parser encounters a token that violates our policy, execution of the current code block can be terminated immediately.

Rendering Engine Besides injecting malicious JavaScript code directly into an application, attackers are able to indirectly trigger the execution of client-side code. For example, the attacker could inject an HTML tag, such as the script or object tag, to make the browser fetch and execute an external script or plugin applet. Hence, only patching the JavaScript engine is not sufficient to prevent DOM-based XSS attacks. To address this issue we additionally patched the HTML parser's logic on how to handle the inclusion of external content. When including active content we again validate the origin of a script's or plugin applet's URL based on our taint information. One possible policy here is to reject URL containing tainted characters. However, as we assess later, real-world applications commonly use tainted data within URLs of dynamically created applets or scripts. Therefore, we allow tainted data within such a remote URL, but we do not allow the tainted data to be contained either in the protocol or the domain of the URL. The only exemption to this rule is the inclusion of external code from the same origin. In these cases, similar to what the Auditor does, we allow the inclusion even if the protocol or domain is tainted. This way, we make sure that active content can only be loaded from hosts trusted by the legitimate Web application.

DOM bindings Very similar to the previous case the execution of remote active content can also be triggered via a direct assignment to a script or object tag's src attribute via the DOM API. This assignment does not take place within the HTML parser but rather inside the DOM API. We therefore patched the DOM bindings to implement the same policy as mentioned above.

Intentional Untainting As our taint-aware browser rejects the generation of code originating from a user-controllable source, we might break cases in which such a generation is desired. A Web application could, for example, thoroughly sanitize the input for later execution. In order to enable such cases we offer an API to taint and untaint strings. If a Web application explicitly wants to opt-out of our protection mechanism, the API can be used to completely remove taint from a string.

5.3 Handling Tainted JSON

While our policy effectively blocks the execution of attacker-injected JavaScript, only allowing literals causes issues with tainted JSON. Although JavaScript provides dedicated functionality to parse JSON, many programmers make use of `eval` to do so. This is mainly due to the fact that `eval` is more tolerant whereas `JSON.parse` accepts only well-formed JSON strings. Using our proposed policy we disallow tokens like braces or colons which are necessary for parsing of JSON. In a preliminary crawl, we found that numerous applications make use of postMessages to exchange JSON objects across origin boundaries. Hence, simply passing on completely tainted JSON to the JavaScript parser would break all these applications whereas allowing the additional tokens to be generated from parsing tainted JSON might jeopardize our protection scheme. In order to combat

these two issues we implemented a separate policy for JSON contained within postMessages. Whenever our implementation encounters a string which heuristically matches the format of JSON, we parse it in a tolerant way and deserialize the resulting object. In doing so, we only taint the data values within the JSON string. This way incompatible Web applications are still able to parse JSON objects via `eval` without triggering a taint exception. Since we validated the JSON's structure, malicious payloads cannot be injected via the JSON syntax. If a deserialized object's attributes are used later to generate code, they are still tainted and attacks can be detected. If for some reason our parser fails, we forward the original, tainted value to the postMessage's recipient to allow for backwards compatibility.

5.4 Implementation

To practically validate the feasibility of our protection approach we conducted a prototypical implementation based on the open source browser, Chromium, version 30.0.1561.0. This section will provide details on a selection of issues we encountered when implementing the desired protection capabilities.

Equality problem for tainted strings: We had to decide when a tainted string should be considered equal to an untainted version as this requirement is dependent on the situation at hand. Under certain circumstances we do want to consider them as being equal but there are also conditions under which equality should not be given. For example, when creating DOM elements from tainted strings, we do want a tainted string to be equal to an untainted version because the tainted string should match the untainted version for the correct element to be created. If the strings would not match, the correct element could not be looked up and hence an unknown (or custom) element would be created. On the other hand, when looking up a string in a cache, we do not want the tainted version to be equal to an untainted one. If that were the case, we might loose taint as we retrieve the untainted version. For performance reasons WebKit uses addresses of certain strings it considers to be unique to perform an equality check. We thus needed to implement a fallback method for the equality check on tainted strings if we desire a tainted string to be equal to its untainted version.

Attaching taint to JavaScript Tokens: To prevent code strings from untrusted sources to generate code, we needed to forward taint information from strings to these generated tokens. We thus needed to broaden the interface not only leading to the JavaScript lexer but also to the parser. V8 not only has a parser for the JavaScript language but also for JSON to efficiently read serialized data. While it was conceptually easy to attach another bit to the generated tokens, a sophisticated buffering logic inside V8 needed to be made taint aware. A variety of `CharacterStream` classes buffer characters of an input stream to be consumed by the scanner and also enables it to push back characters if it did not accept them. To enable taint propagation all classes of an inheritance hierarchy at least three levels deep needed to be changed.

6 Practical Evaluation

After the implementation of our modified engine as well as the augmented HTML and JavaScript parsers we evaluated our approach in three different dimensions. In this section we shed light on the compatibility of our approach with the current Web, discuss its protection capabilities, and evaluate its performance in comparison to the vanilla implementation of Chromium as well as other commonly used browsers. Finally, we summarize the results of said evaluation and discuss their meaning.

6.1 Compatibility

While a secure solution seems desirable, it will not be accepted by users if it negatively affects existing applications. Therefore, in the following, we discuss the compatibility of our proposed defense to real-world applications. We differentiate between the two realms in which our approach is deployed – namely the JavaScript parser and the HTML/DOM components – and answer the questions:

1. In what fraction of the analyzed applications do we break at least one functionality?
2. How big is the fraction of all documents in which we break at least one functionality?
3. How many of these false positives are actually caused by vulnerable pieces of code which allow an attacker to execute a Cross-Site Scripting attack?

6.1.1 Analysis methodology

To answer these questions for a large body of domains, we conducted a shallow crawl of the Alexa Top 10,000 domains (going down one level from the start page) with our implemented filter enabled. Rather than just blocking execution we also sent a report back to our backend each time the execution of code was blocked. Among the information sent to the backend were the URL of the page that triggered the exception, the exact string that was being parsed as well as the corresponding taint information. Since we assume that we are not subject to a DOM-based XSS attack when following the links on said start pages, we initially count all blocked executions of JavaScript as false positives. In total, our crawler visited 981,453 different URLs, consisting of a total of

9,304,036 frames. The percentages in the following are relative to the number of frames we analyzed.

6.1.2 Compatibility of JavaScript Parsing Rules

In total, our crawler encountered and subsequently reported taint exceptions, i.e., violations of the aforementioned policy for tainted tokens, in 5,979 frames. In the next step, we determined the Alexa ranks for all frames which caused exceptions, resulting in a total of 50 domains. Manual analysis of the data showed that on each of these 50 domains, only one JavaScript code snippet was responsible for the violation of our parsing policy. Out of these 50 issues, 23 were caused by data stemming from a postMessage, whereas the remaining 27 could be attributed to data originating from the URL. With respect to the analyzed data set this means that the proposed policy for parsing tainted JavaScript causes issues on 0.50% of the domains we visited, whereas in total only 0.06% of the analyzed frames caused issues.

To get a better insight into whether these false positive were in fact caused by vulnerable JavaScript snippets, we manually tried to exploit the flows which had triggered a parsing violation. Out of the 23 issues related to data from postMessages, we found that one script did not employ proper origin checking, allowing us to exploit the insecure flow. Of the 27 other scripts which were not using data from postMessages, we were able to exploit 21 scripts and hence 21 additional domains. This constitutes a total number of 50 domain on which one functionality caused a false positive, while 22 domains contained an actual vulnerability in just the functionality our filter blocked.

Importance of JSON-handling policy As we outlined in Section 5.3, we do allow for postMessages to contain tainted JSON which is automatically selectively untainted by our prototype. To motivate the necessity for this decision, we initially also gathered taint exceptions caused by tainted JSON (stemming from postMessages) being parsed by `eval`. This analysis showed that next to the 5,979 taint exceptions we had initially encountered, 90,937 additional documents utilized tainted JSON from postMessages in a policy-violating manner. Albeit, with respect to our data set, this only caused issues with less than 1% of all documents we analyzed, it puts emphasis on the necessity for our proposed selective untainting, whereas on the other hand, it also shows that programmers utilize `eval` quite often in conjunction with JSON exchanged via postMessages.

6.1.3 Compatibility of HTML Injection Rules

As discussed in the Section 5.2, our modified browser blocks the execution of external scripts if any character in the domain name of the external script resources is tainted – only exempting those scripts that are located on the same domain as the including document. Analogous to what we had investigated with respect to the JavaScript parsing policy, we wanted to determine in how many applications we would potentially break functionality when employing the proposed HTML parsing policy. We therefore implemented a reporting feature for *any* tainted HTML and a blocking feature for policy-violating HTML. This feature would always send a report containing the URL of the page, the HTML to be parsed, as well as the exact taint information to the backend. We will go into further detail on injected HTML in Section 7 and will now focus on all those tainted HTML snippets which violate the policy we defined in Section 5.2.

During our crawl, 8,805 out of the 9,304,036 documents we visited triggered our policy of tainted HTML, spreading across 73 domains. Out of these, 8,667 violations (on 67 domains) were caused by script elements with `src` attributes containing one or more tainted characters in the domain of the included external script. Out of the remaining six domains, we found that three utilized `base.href` such that the domain name contained tainted characters and thus, our prototype triggered a policy exception on these pages. Additionally, two domains used policy-violating `input.formaction` attributes and the final remaining domain had a tainted domain name in an `embed.src` attribute. In total, this sums up to a false positive rate of 0.09% with respect to documents as well as 0.73% for the analyzed domains.

Subsequently, we analyzed the 73 domains which utilized policy violation HTML injection to determine how many of them were susceptible to a DOM-based XSS attack. In doing so, we found that we could exploit the insecure use of user-provided data in the HTML parser in 60 out of 73 cases.

6.1.4 Compatibility of DOM API Rules

As we discussed previously we also examine assignments to security-critical DOM properties like `script.src` or `base.href` and block them according to our policy. In our compatibility crawl, our engine blocked such assignments on 60 different domains in 182 documents, whereas the largest amount of blocks could be attributed to `script.src`. Noteworthy in this instance is the fact that 45 out of these 60 blocks interfered with third-party advertisement by only two providers.

After having counted the false positive, we yet again tried to exploit the flows that had been flagged as malicious by our policy enforcer. Out of the 60 domains our enforcer had triggered a block on, we verified that eight constitute exploitable vulnerabilities. In compari-

	documents	domains	exploitable domains
JavaScript	5,979	50	22
HTML	8,805	73	60
DOM API	182	60	8
Sum	14,966	183	90

Table 1: False positives by blocking component

son to the amount of exploitable blocks we had encountered for the JavaScript and HTML injection polices this number seems quite low. This is due to the fact that both the aforementioned advertisement providers employed whitelisting to ensure that only script content hosted on their domains could be assigned. In total, this sums up to 0.60% false positives with respect to domains and just 0.002% of all analyzed documents.

6.1.5 Summary

In this section we evaluated the false positive rate of our filter. In total, the filtering rules inside the JavaScript parser, the HTML parser and the security-sensitive DOM APIs caused issues on 14,966 document across 183 domains. Considering the data set we analyzed this amounts to a false positive ratio of 0.16% for all analyzed documents and 1.83% for domains. Noteworthy in this instance is however the fact that out of the 183 domains on which our filter blocked a single functionality, 90 contained actual verified vulnerabilities in just that functionality. Table 1 shows the number of documents and domains on which our policy caused false positive, also denoting in which of the different policy-enforcing components the exception was generated as well as the amount of domains in which the blocked functionality caused an exploitable vulnerability.

6.2 Protection

To ensure that our protection scheme does not perform worse than the original Auditor, we re-ran all exploits that successfully triggered when the Auditor was disabled. All exploits were caught by the JavaScript parser, showing that our scheme is at least as capable of stopping DOM-based Cross-Site Scripting as the Auditor.

To verify the effectiveness of our proposed protection scheme, we ran all generated exploits and bypasses against our newly designed filter. To minimize side-effects, we also disabled the XSS Auditor completely to ensure that blocking would only be conducted by our filtering mechanism. As we discussed in Section 4.2, alongside the scoping-related issues that were responsible for the successful bypassing of the Auditor by the first generation of exploits, other issues related to string matching arose. In the following, we briefly discuss our protection scheme with respect to stopping these kinds of exploits.

Scoping: eval and innerHTML In contrast to the XSS Auditor our filtering approach is fully capable of blocking injections into `eval` due to the fact that it is implemented directly in the JavaScript parser. In the XSS Auditor, innerHTML is not checked for performance reasons. To check whether a given token was generated from a tainted source, a simple boolean flag has to be checked, therefore we do not have these performance-inhibiting issues.

Injection attacks targeting DOM APIs In our experiments, we did not specifically target the direct assignment to security-critical DOM API properties. Inside the API, analogous to the HTML parser, assignment to one of these critical properties might cause direct JavaScript execution (such as a `javascript:` URL for an `iframe.src`) or trigger loading of remote content. For the first case, our taint tracking approach is capable of persisting the taint information to the execution of the JavaScript contained in the URL and hence, the DOM API does not have to intervene. For the loading of remote content, the rules of the HTML parser are applied, disallowing the assignment of the property if the domain name contains tainted characters.

Partial injection One of the biggest issues, namely partial injection, was stopped at multiple points in our filter. Depending on the element and attribute which could be hijacked, the attack vector either consisted of injected JavaScript code or of an URL attribute used to retrieve foreign content (e.g. through `script.src`). For the direct injection of JavaScript code, the previously discussed JavaScript parser was able to stop all exploit prototypes whereas for exploits targeting URL attributes the taint-aware HTML parser successfully detected and removed these elements, thus stopping the exploit.

Trailing content and double injection The bypasses which we categorized as trailing content are targeting a weakness of the Auditor, specifically the fact that it searches for completely injected tags whereas double injection bypasses take advantage of the same issue. Both trailing content and double injections can be abused to either inject JavaScript code or control attributes which download remote script content. Hence, analogous to partial injection, the filtering rules in the HTML and JavaScript parsers could in conjunction with the precise origin information stop all exploits.

Second order flows and alternative attack vectors Similar to injection attacks targeting the direct assignment of DOM properties through JavaScript, we did not generate any exploits for second order flows. Nevertheless, we persist the taint information through intermedi-

ary sources like the WebStorage API. Therefore, our prototype is fully capable of detecting the origin of data from these intermediary source and can thus stop these kinds of exploits as well. As for postMessages, `window.name` and `document.referer`, our implementation taints all these sources of potentially attacker-controlled data and is hence able to stop all injection attacks pertaining to these sources.

Application-specific input mutation Our engine propagates taint information through string modification operations. Therefore, it does not suffer the drawbacks of current implementations based on string matching. All exploits targeting vulnerabilities belonging to this class were caught within our HTML and JavaScript parsers.

6.3 Performance

In order to evaluate the performance of our implementation we conducted experiments with the popular JavaScript benchmark suites Sunspider, Kraken, and Octane as well as the browser benchmark suite Dromaeo. Sunspider was developed by the WebKit authors to "focus on short-running tests [that have] direct relevance to the web" [28]. Google has developed Octane which includes "5 new benchmarks created from full, unaltered, well-known web applications and libraries" [5]. Mozilla has developed Kraken which "focuses on realistic workloads and forward-looking applications" [15]. Dromaeo, which is a combination of several JavaScript and HTML/DOM tests, finally serves as a measure of the overall impact our implementation has on the everyday browsing experience.

All tests ultimately lead to a single numerical value, either being a time needed for a run (the lower the better) or a score (the higher the better), reflecting the performance of the browser under investigation. For runtime (score) values the results were divided by the values obtained for the unmodified version of the Web browser (vice versa). These serve as the baseline for our further comparisons. With the obtained results we computed a slowdown factor reflecting how many times slower our modified version is. To set these numbers into context, we also evaluated other popular Web browsers, namely Internet Explorer 11 and Firefox 26.0. To eliminate side effects of, e.g., the operating system or network latency, we ran each of the benchmarking suites locally for ten times using an Intel Core i7 3770 with 16GB of RAM. All experiments, apart from Internet Explorer, were conducted in a virtual machine running Ubuntu 13.04 64-bit on that system whereas IE was benchmarked natively running Windows 7.

Table 2 shows the results of our experiments. To ascertain a baseline for our measurements we ran all benchmarks on a vanilla build of Chromium. The table shows the mean results (in points or milliseconds) as well as the standard error and the slowdown factor for each test and browser. Internet Explorer employs an optimization to detect and remove dead code, causing it to have significantly better performance under the Sunspider benchmark than the other Web browsers [40]. As the results generated by all browsers under the Kraken benchmark were varying rather strongly, we ran the browsers in our virtual machines 50 times against the Kraken benchmark. Regardless, we still see a rather high standard error of the mean for all the browsers.

We chose the aforementioned tests because they are widely used to evaluate the performance of both JavaScript and HTML rendering engines. Nevertheless, these tests are obviously not designed to perform operations on tainted strings. As we discussed in Section 5.4, our engine usually only switches to this runtime implementation if the operation is conducted on a tainted string. In the initial runs, which is denoted in Table 2 as *Tainted Best*, our engine incurred slowdown factors of 1.08, 1.01, 1.16 and 1.05, resulting in an average slowdown factor of 7%. Since the tests are not targeting the usage of tainted data, we conducted a second run. This time we modified our implementation to treat *all* strings as being tainted, forcing it to use as much of our new logic as possible. In this, the performance was naturally worse than in the first run. More precisely, by calculating the average over the observed slowdown factors for our modified (denoted as *Tainted Worst*) version, we see that our implementation incurs, in the worst case, an overhead of 17% compared to the vanilla version. While the performance hit is significant, we will elaborate on possible performance improvements in the next section.

6.4 Discussion

In this section we evaluated compatibility, protection capability as well as performance of our proposed filter against DOM-based Cross-Site Scripting. In the following we will briefly discuss the implications of these evaluations.

In our compatibility crawl we found that 183 of the 10,000 domains we analyzed had one functionality that was incompatible with our policies for the JavaScript parser, the HTML parser and the DOM APIs. Although this number appears to be quite high at first sight it also includes 90 domains on which we could successfully a vulnerability in just the functionality that was blocked by our filter. On the other hand, the total number of domains, which our approach protected from a DOM-based XSS attack amounts to 958. Although the XSS Auditor is not designed to combat DOM-based XSS attacks, it is the only currently employed defense for Google Chrome against such attacks. As we discussed in Section 4.3, the Auditor could be bypassed on 81% of these domains, protecting users on only 183 domains in our initial data

	Dromaeo			Octane			Kraken (ms)			Sunspider (ms)		
Baseline	1167.4	*1.89*	–	20177.9	*64.47*	–	1418.9	*94.29*	–	169.02	*0.37*	–
Tainted Best	1082.6	*2.40*	**1.08**	19851.0	*54.54*	**1.01**	1653.1	*59.84*	**1.16**	178.03	*0.70*	**1.05**
Tainted Worst	1015.6	*1.93*	**1.15**	18168.7	*70.24*	**1.11**	1814.4	*64.55*	**1.27**	192.66	*0.26*	**1.14**
Firefox 26.0	721.7	*2.94*	**1.62**	16958.5	*97.40*	**1.19**	1291.3	*1.14*	**0.91**	171.86	*0.65*	**1.02**
IE 11	607.0	*2.13*	**1.92**	17247.2	*47.15*	**1.17**	1858.5	*4.16*	**1.31**	78.05	*0.13*	**0.46**

Table 2: Benchmark results, showing mean, *standard error* and **slowdown factor** for all browsers and tests

set. This shows that with respect to its protection capabilities our approach is more reliable than currently deployed techniques, which do not offer protection against this type of attack.

Apart from reliable protection and a low false positive rate, one requirement for a browser-based XSS filter is its performance. Our performance measurements showed that our implementation incurs an overhead between 7 and 17%. Chrome's JavaScript engine V8 draws much of its superior performance from utilizing so-called *generated code*, i.e., ASM code generated directly from macros. To allow for a small margin for error, we opted to implement most of the logic – such as copying of taint information – in C++ runtime code. We realize that the performance impact of our current prototype might be too high to allow for productive deployment in Chrome. Nevertheless, we believe that an optimized implementation making more frequent use of said generated code would ensure better performance and possibly allow for deployment of our approach.

Our approach only aims at defeating DOM-based Cross-Site Scripting while the XSS Auditor's focus is on reflected XSS. We therefore believe that deployment besides the Auditor is a sound way of implementing a more robust client-side XSS filter, capable of handling both reflected and DOM-based XSS.

7 Outlook: HTML Injection

As we discussed in Section 5.4, our engine allows for precise tracking of tainted data throughout the execution of a program and hence, also to the HTML parser. Therefore, our approach also enables the browser to precisely block all attacker-injected HTML even it is not related to Cross-Site Scripting. Although this was out of scope for this work, we believe that it is relevant future work. Therefore, we give a short glimpse into the current state of the Web with respect to partially tainted HTML passed to the parser.

As we discussed in Section 6.1, we conducted a compatibility crawl of the Alexa Top 10,000 in which we analyzed a total of 9,304,036 documents, out of which 632,109 generated 2,393,739 tainted HTML reports. Typically, each of the HTML snippets contained the definition of more than one tag. In total, we found that parsing the snippets yielded in 3,650,506 tainted HTML elements whereas we consider an element tainted if either the tag name, any attribute name or any attribute value is tainted. Considering the severity of attacker-controllable HTML snippets, we distinguish between four types:

1. Tag injection (**TI**): the adversary can inject a tag with a name of his choosing.
2. Attribute injection (**AI**): injection of the complete attribute, namely both name and value
3. Full attribute value injection (**FAVI**): full control over the value, but not the name
4. Partial attribute value injection (**PAVI**): attacker only controls part of the attribute

We analyzed the data we gathered in our crawl to determine whether blocking of tainted HTML data is feasible and if so, with what policy. Our analysis showed that out of the Top 10,000 Alexa domains, just one made use of full tag injection, injecting a `p` tag originating from a postMessage. This leads us to believe that full tag injection with tainted data is very rare and not common practice.

The analysis also unveiled that the most frequently tainted elements – namely `a`, `script`, `iframe` and `img` – made up for 3,503,655 and thus over 95% of all elements containing any tainted data. Hence, we focused our analysis on these and examined which attributes were tainted. Analogous to our definition of a tainted element, we consider an attribute to be tainted if either its name or value contains any tainted characters. Considering this notion, we – for each of the four elements – ascertained which attribute is most frequently injected using tainted data. For `a` elements, the most frequent attribute containing tainted data was `href` whereas `script`, `iframe` and `img` tags mostly had tainted `src` attributes. Although we found no case where the name of an attribute was tainted, we found a larger number of elements with full attribute value injection. The results of our study are depicted in Table 3, which shows the absolute numbers of occurrences. We also gathered reports from documents

	FAVI		PAVI	
	Top 10k	all	Top 10k	all
iframe.src	349	2,222	384,946	438,415
script.src	4,215	8,667	1,078,015	1,292,046
a.href	124,812	133,838	1,162,093	1,191,598
img.src	5,128	6,791	275,579	312,033
Domains	799	1,014	4,446	6,772

Table 3: Amounts of full and partial value injection for domains in the Alexa Top 10,000 and beyond.

on domains not belonging to the Alexa Top 10,000 as content is often included from those. The first number in each column gives the amount for documents on the Alexa Top 10,000, whereas the second number shows the number for all documents we crawled.

Summarizing, we ascertain that taint tracking in the browser can also be used to stop HTML injection. Our study on tainted HTML content on the Alexa Top 10,000 domains has shown that blocking elements with tainted tag names is a viable way of providing additional security against attacks like information exfiltration [7] while causing just one incompatibly. We also discovered that the applications we crawled do not make use of tainted attribute names, hence we assume that blocking tainted attributes does also not cause incompatibilities with the current Web. In contrast, blocking HTML that either has fully or partially tainted attribute values does not appear to be feasible since our analysis showed that 8% of all domains make use of fully tainted attribute values whereas more than 44% used partially tainted values in their element's attributes. As there is an overlap between these two groups of domains, the total number of domains that would causes incompatibilities is 4,622, thus resulting in more than 46% incompatibilities. Thus, we established that although blocking HTML is technically possible with our implementation this would most likely break a large number of applications.

8 Related work

XSS Filter As already mentioned earlier, the conceptually closest work to this paper is Bates et al.'s [2] analysis of regular expression-based XSS filters and the subsequent proposal of the methodology that constitutes the basis for the XSS Auditor. Furthermore, Pelizzi and Sekar [26] proposed potential improvements for Bates et al.'s method in order to address the problem of partial injections. Similar to what Bates et al. discussed, they instrument the HTML parser and apply approximate string matching inside it. Due to the fact that DOM-based XSS allows an attacker to make use of insecure calls to `eval` as well as direct assignments to security-sensitive DOM APIs, it is still susceptible to some bypasses we discussed. Furthermore, the presented approach is not fully evaluated, especially with respect to the occurring false positive rate. Besides this, and the other two major browser-based XSS filters [21, 29], the majority of XSS protection approaches, such as [23, 14, 24, 35], reside on the server-side.

Filter evasion is an active topic especially in the offensive community. Academic approaches in this area include, for instance, the work by Heiderich et al. on SVG-based evasions [11] and filter evasion by misusing browser-based parser quirks and mutations [12] as well as approaches that rely on parser confusion and polyglots, such as Barth et al. [1] and Magazinius et al. [19].

Dynamic taint tracking Taint propagation is a well established tool to address injection attacks. After its initial introduction within the Perl interpreter [37], various server-side approaches have been presented that rely on this technique [25, 27, 33, 24, 3]. In 2007, Vogt et al. [36] pioneered browser-based dynamic taint tracking, employing the technique to prevent the leakage of sensitive data to a remote attacker rather than trying to prevent the attack itself. The first work to utilize taint tracking for the detection of DOM-based XSS was DOMinator [8], which was later followed by FLAX [31] and Lekies et al. [18]. For NDSS 2009, Sekar [32] proposed and implemented a scheme for taint inference, speeding up taint tracking approaches which had been presented up to this point.

9 Conclusion

In this paper we presented the design, implementation and thorough evaluation of a client-side countermeasure which is capable to precisely and robustly stop DOM-based XSS attacks. Our mechanism relies on the combination of a taint-enhanced JavaScript engine and taint-aware parsers which block the parsing of attacker-controlled syntactic content. Existing measures, such as the XSS Auditor, are still valuable to combat XSS in cases that are out of scope of our approach, namely XSS which is caused by vulnerable data flows that traverse the server.

In case of client-side vulnerabilities, our approach reliably and precisely detects injected syntactic content and, thus, is superior in blocking DOM-based XSS. Although our current implementation induces a runtime overhead between 7 and 17%, we believe that an efficient native integration of our approach is feasible. If adopted, our technique would effectively lead to an extinction of DOM-based XSS and, thus, significantly improve the security properties of the Web browser overall.

Acknowledgment

This work was in parts supported by the EU Projects WebSand (FP7-256964) and STREWS (FP7-318097). The support is gratefully acknowledged. The authors would also like to thank Sven Kälber for providing the computational power to conduct our data collection as well as the anonymous reviewers and our shepherd Adrienne Porter Felt for their help comments and support.

References

[1] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 360–371. IEEE, 2009.

[2] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[3] Prithvi Bisht and VN Venkatakrishnan. Xss-guard: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2008.

[4] Hristo Bojinov, Elie Bursztein, and Dan Boneh. Xcs: cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009.

[5] Stefano Cazzulani. Octane: the javascript benchmark suite for the modern web. Online, `http://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html`, August 2012.

[6] CERT. Advisory ca-2000-02 malicious html tags embedded in client web requests. [online], `http://www.cert.org/advisories/CA-2000-02.html`, February 2000.

[7] Eric Y Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Proceedings of the Workshop of Web*, volume 2. Citeseer, 2012.

[8] Stefano Di Paola. DominatorPro: Securing Next Generation of Web Applications. [online], `https://dominator.mindedsecurity.com/`, 2012.

[9] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1065–1074. ACM, 2008.

[10] Steve Hanna, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The emperors new apis: On the (in) secure usage of new client-side primitives. In *Proceedings of the Web*, volume 2, 2010.

[11] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. Crouching tiger-hidden payload: security risks of scalable vectors graphics. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 239–250. ACM, 2011.

[12] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. mxss attacks: attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 777–788. ACM, 2013.

[13] Cormac Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 workshop on New security paradigms workshop*, pages 133–144. ACM, 2009.

[14] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 145–151. IEEE, 2004.

[15] Erica Jostedt. Release the kraken. Online, `https://blog.mozilla.org/blog/2010/09/14/release-the-kraken-2/`, Sept. 2010.

[16] Amit Klein. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles*, 4, 2005.

[17] Andreas Kling. Xssauditor performance regression due to threaded parser changes. [online], `https://gitorious.org/webkit/webkit/commit/aaad2bd7c86f78fe66a4c709192e3b591c557e7a`, April 2013.

[18] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204. ACM, 2013.

[19] Jonas Magazinius, Billy K Rios, and Andrei Sabelfeld. Polyglots: crossing origins by crossing formats. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 753–764. ACM, 2013.

[20] Georgio Maone. Noscript. [online], `http://www.noscript.net/whats`.

[21] Georgio Maone. Noscripts anti-xss protection. [online], `http://noscript.net/featuresxss`.

[22] Georgio Maone. Noscripts anti-xss filters partially ported to ie8. [online], `http://hackademix.net/2008/07/03/noscripts-anti-xss-filters-partially-ported-to-ie8/`, July 2008.

[23] Raymond Mui and Phyllis Frankl. Preventing web application injections with complementary character coding. In *Computer Security–ESORICS 2011*, pages 80–99. Springer, 2011.

[24] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[25] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.

[26] Riccardo Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *AsiaCCS*, May 2012.

[27] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection*, RAID'05, pages 124–145, Berlin, Heidelberg, 2006. Springer-Verlag.

[28] Filip Pizlo. Announcing sunspider 1.0. [online], `https://www.webkit.org/blog/2364/announcing-sunspider-1-0/`, April 2013.

[29] David Ross. IE 8 XSS Filter Architecture / Implementation. [online], `http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx`, August 2008.

[30] David Ross. IE8 Security Part IV: The XSS Filter. [online], `http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx`, July 2008.

[31] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[32] R Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[33] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[34] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, pages 399–416, 2009.

[35] Mike Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE, 2009.

[36] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[37] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[38] Web Hypertext Application Technology Working Group. Cross-document messaging. Online, `http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html`.

[39] WhiteHat Security. Website security statistics report. [online], `https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf`, May 2013.

[40] Windows Internet Explorer Engineering. Html5, and real world site performance: Seventh ie9 platform preview available for developers. Online, `http://blogs.msdn.com/b/ie/archive/2010/11/17/html5-and-real-world-site-performance-seventh-ie9-platform-preview-available-for-developers.aspx`.

On the Effective Prevention of TLS Man-In-The-Middle Attacks in Web Applications

Nikolaos Karapanos and Srdjan Capkun
Department of Computer Science, ETH Zurich
{*firstname.lastname*}*@inf.ethz.ch*

Abstract

In this paper we consider TLS Man-In-The-Middle (MITM) attacks in the context of web applications, where the attacker is able to successfully impersonate the legitimate server to the user, with the goal of impersonating the user to the server and thus compromising the user's online account and data. We describe in detail why the recently proposed client authentication protocols based on TLS Channel IDs, as well as client web authentication in general, cannot fully prevent such attacks.

Nevertheless, we show that strong client authentication, such as Channel ID-based authentication, can be combined with the concept of server invariance, a weaker and easier to achieve property than server authentication, in order to protect against the considered attacks. We specifically leverage Channel ID-based authentication in combination with server invariance to create a novel mechanism that we call SISCA: Server Invariance with Strong Client Authentication. SISCA resists user impersonation via TLS MITM attacks, regardless of how the attacker is able to successfully achieve server impersonation. We analyze our proposal and show how it can be integrated in today's web infrastructure.

1 Introduction

Web applications increasingly employ the TLS protocol to secure HTTP communication (i.e., HTTP over TLS, or HTTPS) between a user's browser and the web server. TLS enables users to securely access and interact with their online accounts, and protects, among other things, common user authentication credentials, such as passwords and cookies. Such credentials are considered *weak*; they are transmitted over the network and are susceptible to theft and abuse, unless protected by TLS.

Nevertheless, during TLS connection establishment, it is essential that the server's authenticity is verified. If an attacker successfully impersonates the server to the user, she is then able to steal the user's credentials and subsequently use them to impersonate the user to the legitimate server. This way, the attacker gains access to the user's account and data which can be abused for a variety of purposes, such as spying on the user [18, 48]. This attack is known as TLS Man-In-The-Middle (MITM).

TLS server authentication is commonly achieved through the use of X.509 server certificates. A server certificate binds a public key to the identity of a server, designating that this server holds the corresponding private key. The browser accepts a certificate if it bears the signature of any trusted Certificate Authority (CA). Browsers are typically configured to trust hundreds of CAs.

An attacker can thus successfully impersonate a legitimate server to the browser by presenting a valid certificate for that server, as long as she holds the corresponding private key. In previous years, quite a few incidents involving mis-issued certificates [2, 9, 11, 48, 49] were made public. Even in the case where the attacker simply presents an invalid (e.g., self-signed) certificate not accepted by the browser, she will still succeed in her attack if the user defies the browser's security warning.

In order to thwart such attacks, various proposals have emerged. Some proposals focus on enhancing the certificate authentication model. Their objective is to prevent an attacker possessing a mis-issued, yet valid certificate, from impersonating the server (e.g., [20, 33, 36, 52]).

Other proposals focus on strengthening client authentication. *Strong* client authentication prevents user credential theft or renders it useless, even if the attacker can successfully impersonate the server to the user. One such prominent proposal is Channel ID-based client authentication, introduced in 2012. TLS Channel IDs [4] are experimentally supported in Google Chrome and are planned to be used in the second factor authentication standard U2F, proposed by the FIDO alliance [22].

In this paper we show that Channel ID-based approaches, as well as web authentication solutions that focus solely on client authentication are vulnerable to an attack that we call *Man-In-The-Middle-Script-In-The-Browser (MITM-SITB)*, and is similar to *dynamic pharming* [32] (see Section 4). This attack bypasses Channel ID-based defenses by shipping malicious JavaScript to the user's browser within a TLS connection with the attacker, and using this JavaScript in direct connections

with the legitimate server to attack the user's account.

Nevertheless, we show that TLS MITM attacks where the attacker's goal is user impersonation can still be prevented by strong client authentication, such as Channel ID-based authentication, provided that it is combined with the concept of *server invariance*, that is, the requirement that the client keeps communicating with the same entity (either the legitimate server, or the attacker) across multiple connections intended for the same server. Server invariance is a weaker requirement than server authentication, and thus, it is easier to achieve as no initial trust is necessary. Building on this observation, we propose a solution called *SISCA: Server Invariance with Strong Client Authentication*, that combines Channel ID-based client authentication and server invariance.

SISCA can resist TLS MITM attacks that are based on mis-issued valid certificates, as well as invalid certificates, requiring no user involvement in the detection of the attack (i.e., no by-passable security warnings when server invariance violation occurs). SISCA also thwarts attackers that hold the private key of the legitimate server.

Contributions. In this work we analyze TLS MITM attacks whose goal is user impersonation and make the following contributions. *(i)* We show, by launching a MITM-SITB attack, that Channel ID-based client authentication solutions do not fully prevent TLS MITM attacks; *(ii)* we further argue that effective prevention of MITM-based user impersonation attacks requires strong user authentication and (at least) server invariance; *(iii)* we propose a novel solution that prevents MITM-based user impersonation, based on the combination of strong client authentication and server invariance; *(iv)* we implement and evaluate a basic prototype of our solution.

2 Channel ID-based Authentication and MITM Attacks

2.1 Attacker Model and Goals

Attacker Goals. The attacker's goal in a MITM attack is typically to impersonate the user (victim) to the legitimate server (e.g., a social networking, webmail, or e-banking website) in order to compromise the user's online account and data. This is indeed the case where the attacker wishes for example to spy on the user [18, 48], or abuse his account for nefarious purposes, e.g., perform fraudulent financial transactions. Alternatively, the attacker could aim to only impersonate the server to the user (and not the user to the server), such that she serves the user with fake content (e.g., fake news). In this paper, we focus on the first, more impactful, scenario.

Attacker Model. We adopt the attacker model considered by Channel IDs [4]. The adversary is able to position herself suitably on the network and perform a TLS MITM attack between the user and the target web server. In other words, the attacker is able to successfully impersonate the server to the user. We distinguish between two types of MITM[1] attackers.

The *MITM+certificate* attacker holds *(i)* a *valid* certificate for the domain of the target web server, binding the identity of the server to the public key, of which she holds the corresponding private key. The attacker, however, has no access to the private key of the target web server. This, for example, can happen if the attacker compromises a CA or is able to force a CA issue such a certificate. Such attacks have been reported in the recent years [2, 9, 11, 48]. Moreover, in this category we also consider a weaker attacker that only holds *(ii)* an *invalid* (e.g., self-signed) certificate. In this case, the attacker will still succeed in impersonating the server to the user if the latter ignores the security warnings of the browser[2], which is a common phenomenon [51].

The *MITM+key* attacker holds the *private key of the legitimate server*. While we are not aware of publicized incidents involving server key compromise, such attacks are feasible, as the Heartbleed vulnerability in OpenSSL has shown [1], and can be very stealthy, remaining unnoticed. Thus, they are well worth addressing [28, 30, 35].

From the above it follows that the attacker is able to obtain the user's weak credentials, namely passwords and HTTP cookies. She is not, however, able to compromise the user's browser or his devices (e.g., mobile phones).

2.2 TLS Channel IDs

Channel IDs is a recent proposal for strengthening client authentication. It is a TLS extension, originally proposed in [15] as *Origin-Bound Certificates* (OBCs). A refined version has been submitted as an IETF Internet-Draft [4]. Currently, Channel IDs are experimentally supported by Google's Chrome browser and Google servers.

In brief, when the browser visits a TLS-enabled web server for the first time, it creates a new private/public key pair (on-the-fly and without any user interaction) and proves possession of the private key, during the TLS handshake. This TLS connection is subsequently identified by the corresponding public key, which is called the Channel ID. Upon subsequent TLS connections to the same web server, or more precisely, to the same web origin, the browser uses the same Channel ID. This enables the web server to identify the same browser across multiple TLS connections.

2.2.1 Channel ID-Based Authentication

By *Channel ID-based authentication* we refer to the use of Channel IDs *throughout* the user authentication process, designed to thwart both types of MITM attackers presented in Section 2.1 [4, §6], [13, §3].

[1] We use the terms "TLS MITM" and "MITM" interchangeably.

[2] We use the term "browser" to refer to any "user agent" in general.

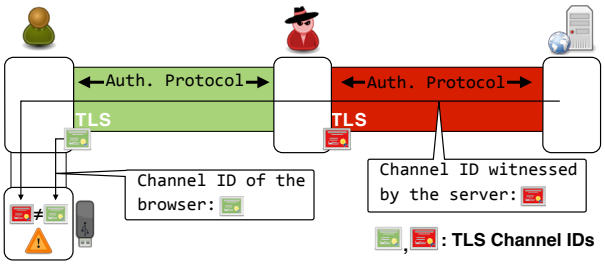


Figure 1: PhoneAuth and FIDO U2F; leveraging Channel IDs to secure the initial login against MITM attacks.

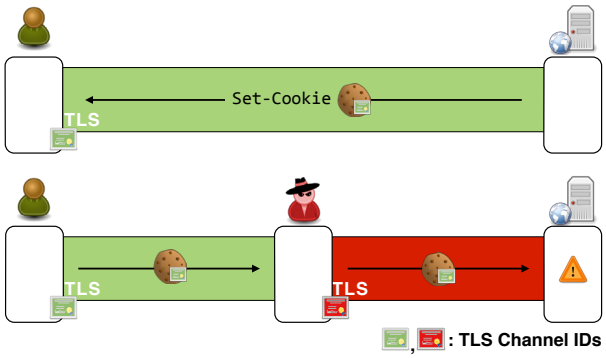


Figure 2: Binding authentication tokens (e.g., cookies) to the browser's Channel ID (green). A MITM attacker who steals such a cookie, cannot use it to impersonate the user, since the attacker has a different Channel ID (red).

Initial Login. When the user attempts to login to his online account for the first time from a particular browser, the web server requires that the user authenticates using a strong second factor authentication device, as in *PhoneAuth* [13] and *FIDO Universal 2nd Factor (U2F)* [22] protocols. These protocols leverage Channel IDs to secure the intial login process against MITM attacks. In brief, as part of the authentication protocol, the second factor device compares the Channel ID of the browser to the Channel ID of the TLS connection that the server witnesses. If they are equal, then the browser is directly connected to the web server through TLS (because they share the same view of the connection), and thus there is no MITM attack taking place. On the other hand, if the Channel IDs differ, then the server is not directly connected to the user's browser. Instead, as shown in Figure 1, there is an attacker in the middle, and the device aborts the authentication protocol, stopping the attack.

Subsequent Logins. Upon successful initial authentication the server sets a cookie to the user's browser, and binds it to the Channel ID of the browser. As proposed in [15], a server may create a *channel-bound cookie* as follows: $\langle v, \text{HMAC}(k, v|cid)\rangle$, where v is the original cookie value, cid is the browser Channel ID and k is a secret key only known to the server, used for computing a MAC over the concatenation of v and cid. The channel-bound cookie is considered valid only if it is presented over that particular Channel ID. Therefore, subsequent interaction with the server from that particular browser is protected by the channel-bound cookie. An attacker that manages to steal a channel-bound cookie, e.g., through a MITM attack, cannot use it to impersonate the user to the web server, since she does not know the private key of the correct Channel ID. Figure 2 illustrates this concept. Note that at this stage, the second factor device is not required for authenticating the user [12].

2.3 MITM Attack on Channel ID-Based Authentication

We show how Channel ID-based authentication still allows a MITM attacker to successfully impersonate the user. This is due to the way web applications are run and interact with the servers, which differs from other internet client-server protocols (e.g., IMAP over TLS).

In particular, web servers are allowed to send scripting code to the browser, which the latter executes within the security context of the web application (according to the rules defined by the *same-origin policy* [5]). In fact, client-side scripting and especially JavaScript, is the foundation of dynamic, rich web applications that vastly improve user experience, and its presence is ubiquitous.

Moreover, a browser can establish multiple TLS connections with the same server. In addition, a typical web application loads resources, such as images and scripts, from multiple domains (*cross-origin* network access [5]). Assuming that all communication is TLS-protected, this means that the browser needs to establish TLS connections with multiple servers while loading a web page.

Given the above, there is a conceptually simple attack that a MITM+certificate or MITM+key attacker can perform, which bypasses the security offered by Channel IDs. We assume that the user tries to access the target web server, say `www.example.com`. The attacker then proceeds as follows:

1. She intercepts a single TLS connection attempt made by the browser to `www.example.com`, and by presenting a valid certificate (or invalid with the user ignoring the browser's warning), she successfully impersonates the legitimate server to the browser.

2. Through the established connection, the browser makes an HTTP request to the server. The attacker replies with an HTTP response, which includes a malicious piece of JavaScript code. This script will execute within the origin of `www.example.com`.

3. The attacker closes the intercepted TLS connection. This forces the browser to initiate a new TLS con-

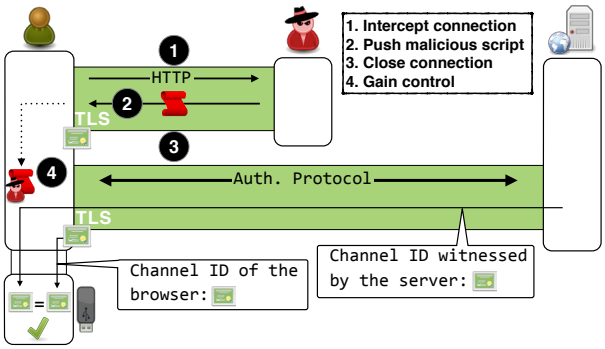


Figure 3: MITM-SITB attack on Channel ID-based PhoneAuth/U2F, used for the initial login. The attacker's JavaScript code is executed within the origin of the target server (shown by the dotted arrow).

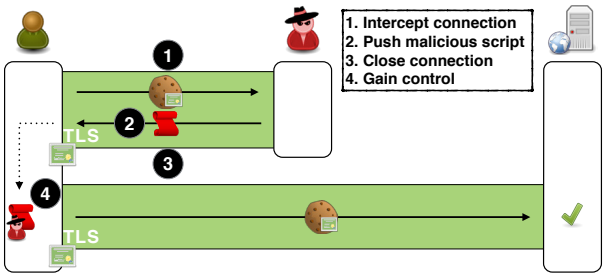


Figure 4: MITM-SITB attack on Channel ID-based authentication after the initial login, where requests are protected with a channel-bound cookie.

nection in order to transmit subsequent requests, or use another existing one, if any (this behavior conforms with the HTTP specification [23]). At the same time, the attacker allows subsequent TLS connection attempts to pass through, without interfering with them. As a result, once the attacker closes that single intercepted connection, all other connections, existing and new, are directly established between the browser and the legitimate server.

4. The attacker gains full control over the user's session in that particular web application. Her script has unrestricted access over the web documents belonging to `www.example.com` and can monitor all the client-side activity of the web application. Moreover, she can issue arbitrary malicious requests to the target server using the `XMLHttpRequest` object [3], in order to perform a desired action or extract sensitive user information. The malicious code can upload any extracted data to an attacker-controlled server. As another example, if the web application is Ajax-based, the attacker can perform *Prototype Hijacking* [46]. This allows her to eavesdrop and modify on-the-fly all the HTTP requests made through `XMLHttpRequest`.

In summary, the MITM attacker "transfers" herself (via the malicious script) within the user's browser, and continues her attack from there. We call this attack *Man-In-The-Middle-Script-In-The-Browser* (*MITM-SITB*).

Figure 3 illustrates the MITM-SITB attack in the case when the user is about to initially authenticate to `www.example.com` using PhoneAuth or U2F. The attacker intercepts a TLS connection, pushes her JavaScript code to the user's browser, and terminates the connection. The browser then establishes a new TLS connection for subsequent communication, only this time with the legitimate server; the attacker will not hijack it. This ensures that the user authentication is performed over a direct connection between the browser and the server, but with the attacker's code running in the browser. The view of the TLS channel will be the same for the browser and the server, and the Channel ID comparison made by the second factor device will pass.

Figure 4 shows how the attack works in the case when the user has already logged in on `www.example.com` in the past, and the server has set a channel-bound cookie in the user's browser. Like before, the attacker ships malicious JavaScript code to the browser by intercepting a TLS connection to `www.example.com`. She then terminates the intercepted connection. This forces the browser to establish a new TLS connection, which is not intercepted by the attacker. This ensures that any subsequent requests, either legitimate or malicious (issued by the attacker's script) are accepted by the legitimate server, since they will carry the channel-bound cookie, which authenticates the user, over the correct Channel ID.

From the above attack description there are various details that remain unclear. For example, which TLS connection the attacker should intercept, whether to "hit and run" or persist as much as possible, etc. Depending on the scenario, there are various alternatives, which are mostly implementation decisions. The attacker can for example choose the following strategy. She intercepts the *very first* TLS connection, i.e., the one that the browser initiates once it is directed to `www.example.com`. Depending on the situation, the attacker's HTTP response could contain the expected HTML document of the website's starting page, together with the appropriately injected malicious script, or it could only contain the malicious script, which will take care of loading the starting page in the browser. Then, as described before, the attacker closes this first connection and subsequent communication (malicious or not) takes place through a direct connection to the legitimate server.

The Cross-Origin Communication Case. Visiting a single web page typically involves cross-origin communication with different domains in the background. For exam-

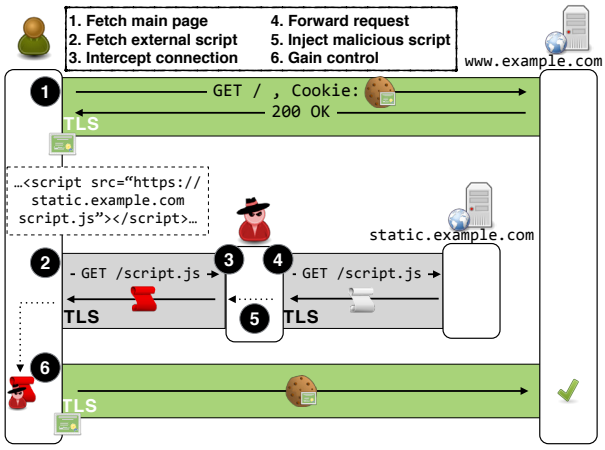


Figure 5: MITM-SITB attack on Channel ID-based authentication leveraging cross-origin communication. Channel IDs for `static.example.com` are of no use.

ple a typical network optimization technique is to have the browser load the static resources of the website, such as images, style sheets and scripts, from so-called *cookieless domains* (e.g., Google websites usually load static resources from `gstatic.com` [24]). These domains, as their name suggests, do not set any cookies, so as to minimize network latency. As a matter of fact, on such domains, client authentication does not apply at all, as they are just used to serve static resources, which anyone, including the attacker, can access. Hence in those cases, the attacker can perform a conventional MITM attack against a cookieless domain, and inject her malicious code at the moment when the target web server requests a legitimate JavaScript file from that domain (Figure 5).

2.4 Proof of Concept Attack

We validate our attack against Channel IDs through a proof of concept implementation. We use two Apache TLS-enabled servers (one for the attacker, one for the legitimate server) and an interception proxy that can selectively forward TLS connections to either server. The legitimate server uses a patched OpenSSL version that supports Channel IDs and leverages them for creating channel-bound cookies. We use Google Chrome as the user's browser, since it supports Channel IDs, and ensure that it accepts the certificates of both servers. We are then able to inject JavaScript code to the user's browser from the attacker's server and issue HTTP requests that are accepted and processed by the legitimate server.

2.5 Scope and Implications of the Attack

The MITM-SITB attack presented in Section 2.3 is not specific to Channel ID-based client authentication protocols. In fact, it applies to *any* web client authentication method. This attack demonstrates that, in the context of web applications, it does not seem possible to prevent TLS MITM attacks via client authentication alone.

We provide the following informal reasoning for the above claim. Client authentication does not prevent an attacker from impersonating the legitimate server. This allows her to intercept a server-authenticated (i.e., TLS) connection and ship her JavaScript code to the user's browser. The browser, treating the attacker's code as trusted (as it came through a server-authenticated connection), executes it within the target server's origin. The attacker accesses the user's account through requests initiated by her code and transmitted over another, direct connection between the browser and the legitimate server.

As a result, schemes such as traditional TLS client authentication [14] and TLS Session Aware User Authentication [42, 43] are still susceptible to TLS MITM attacks, via MITM-SITB. The attacker succeeds in impersonating the user to the web server and compromising his account.

3 Addressing TLS MITM Attacks

As shown in Section 2, strong client authentication alone is not sufficient to prevent MITM attacks that lead to user impersonation in web applications. So, how can we effectively prevent such attacks? In this section we show that there are two *orthogonal* solutions; *(i)* the known solution of preventing the attacker from impersonating the legitimate server at all, i.e., ensuring correct server authentication; *(ii)* our novel approach of combining strong client authentication with server invariance.

3.1 Prevent Server Impersonation

The known and straightforward solution to the problem at hand is to prevent the attacker from impersonating the server in the first place. This way, the attacker can neither steal weak user credentials in order to mount a conventional MITM attack, nor ship malicious Javascript in order to mount a MITM-SITB attack. Note that in this case, strong client authentication (e.g., Channel ID-based) is not necessary for preventing MITM attacks (it is, however, still useful for preventing other attacks, such as phishing and server password database compromise).

The solutions that try to prevent server impersonation essentially address the issue of forged server certificates (and thus defeating MITM+certificate attacks), by performing *enhanced certificate verification*. Such solutions are mainly based on *pinning* [20, 38], *multi-path probing* [33, 36, 37, 52] and hybrid approaches [19, 29] (a thorough survey can be found in [10]).

3.2 Our Proposal: SISCA

3.2.1 Main Concept

The fact that strong client authentication alone cannot effectively prevent MITM attacks in web applications,

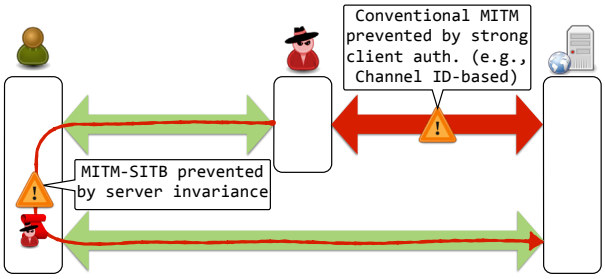


Figure 6: TLS MITM attacks in web applications can be thwarted by combining strong client authentication with server invariance.

raises the following question. Is there a way to somehow still benefit from strong client authentication with respect to addressing MITM attacks?

To answer, we make the following observation. In the context of web applications, a MITM attacker can perform user impersonation via two approaches:

1. The *conventional MITM* attack, in which the attacker compromises the user's credentials and uses them for impersonation. This attack can be effectively prevented by strong client authentication e.g., using Channel ID-based protocols (Figures 1, 2).

2. The *MITM-SITB* attack, presented in Section 2.3 (Figures 3, 4, 5). As discussed in Section 2.5, client authentication alone cannot prevent this attack.

For the MITM-SITB attack to be successful, the user's browser needs to communicate with two different entities, namely the attacker and the target web server. Communicating with the attacker is, of course, necessary for injecting the attacker's script to the browser through the intercepted TLS connection. In addition, communication with the target server is essential, so that the attacker accesses the user's account and data, through her script.

As a result, we can prevent MITM-SITB by making sure that the browser communicates only with *one entity*, either the legitimate server, or the attacker, but *not* with both, during a *browsing session* (a browsing session is terminated when the user closes the browser). In other words, we need to enforce server invariance. When combined with strong client authentication (e.g., Channel ID-based), which stops the conventional MITM approach, this technique manages to effectively thwart MITM attacks. Figure 6 illustrates the concept.

In the remaining section we present a novel solution, called *Server Invariance with Strong Client Authentication (SISCA)*, which stems from the above result. SISCA is able to resist MITM+certificate attacks, offering advantages compared to existing solutions that focus at preventing server impersonation (see Section 3.2.9), as well as MITM+key attacks under the assumption that the attacker does not persistently compromise the server (see Section 3.2.2). The details of our solution follow below.

3.2.2 Design Goals and Assumptions

In SISCA we seek to satisfy the following requirements: *(i)* incremental deployment, *(ii)* scalability, *(iii)* minimal overhead, *(iv)* account for cross-origin communication, assuming that the involved origins belong to, and are administered by the *same entity*, *(v)* mitigation of MITM+key attacks (besides MITM+certificate attacks).

We make the following assumptions. First, strong client authentication, which prevents the conventional way of implementing MITM attacks (Figures 1, 2) is in place. Specifically, we assume that SISCA-enabled servers implement *Channel ID-based client authentication*. As mentioned before, Channel IDs are already experimentally supported in Google Chrome. Moreover, FIDO U2F leverages Channel IDs, as metiononed in Section 2.2.1, so it is likely that Channel ID-based authentication will become available in the foreseeable future.

Second, we assume that SISCA-enabled servers support *TLS with forward secrecy* by default [28, 30, 35]. As we discuss below, this is only required for preventing MITM+key attacks (not relevant for MITM+certificate attacks). Moreover, we assume that TLS is secure and cannot be broken by cryptographic attacks, such as those surveyed in [10].

We finally assume that the MITM+key attacker does not persistently compromise the target web server. As we discuss later, this enables SISCA to resist server key compromise (i.e., MITM+key attackers) through frequent rotation of the server secrets that are used in SISCA (see Section 3.2.8). We also note that if an attacker gained persistent control over the target server, she would probably not need to resort to MITM attacks to compromise the users' accounts, but at the same time she would increase the probability of being detected.

3.2.3 Server Invariance Versus Authentication

As stated above, our goal is to combine strong client authentication with server invariance. Invariance is a *weaker* property than authentication, and thus, *easier* to achieve, as *no a priori trust* is necessary. In contrast, authentication requires some form of initial trust so that the client can correctly authenticate the server [17].

Consequently, we stress the following very important difference. Server authentication (and solutions that try to enforce it, like those mentioned in Section 3.1) implies that every single TLS connection should be established with the legitimate server. If the attacker attempts to intercept such a connection, she should be detected by the browser, i.e., no server impersonation should be possible.

In contrast, server invariance, embraces the fact that

the attacker can successfully impersonate the server. As such, we distinguish two scenarios concerning the browser's *first connection* to a particular server: *(i)* The first connection is *not intercepted* by the attacker. Then, server invariance implies that the attacker is allowed to *intercept none* of the subsequent connections to that server. *(ii)* The first connection is *intercepted* by the attacker. Then, server invariance implies that the attacker has to *intercept all* subsequent connections to that server. In either scenario, if the attacker violates server invariance, she will be detected.

We consider server invariance as a *transient* property whose scope is one browsing session. Server invariance is *reset* whenever the browser restarts, i.e., the attacker is allowed again to choose whether to intercept or not the first connection to the server.

3.2.4 Towards Implementing Server Invariance

In order to implement server invariance, it is important to understand the implications of the fact that the attacker is allowed to impersonate the server. Namely, the attacker can intercept the first connection and influence the entire HTTP response, which clearly cannot be blindly trusted. Therefore, techniques that assume the attacker is able to influence only a part of the HTTP response, such as *Content Security Policy* (CSP) [50] for mitigating *Cross-Site-Scripting* (XSS) [44], as well as techniques that assume the first connection is trusted (i.e., not intercepted by the attacker), such as pinning, cannot be directly applied for implementing server invariance.

Instead, a server invariance protocol should consist of *two phases*, namely *invariance initialization* and *invariance verification* – initialization and verification for brevity. In the initialization phase, which is executed in the first connection to the server during a browsing session (and could be intercepted by the attacker), the browser establishes a *point of reference*. Then, in subsequent connections to the same server, the verification phase is executed, where the browser verifies that the point of reference remains unchanged, i.e., the browser keeps connecting to the same entity.

Server Public Keys. Assuming that we only consider MITM+certificate attackers, we can leverage the servers' public keys as the point of reference. Even if the attacker intercepts the first connection, she will not be able to let any subsequent connections reach the legitimate server, because the server's public key will be different from the attacker's. Nevertheless, servers of the same domain may use different public keys and also, cross-origin interacting domains will have different keys. To solve this issue, we need to "tie" all the involved public keys together, to reflect the fact that they belong to the same entity and thus server invariance should hold across all these domains and keys. We sketch the following technique for implementing server invariance.

During initialization (first connection), the server sends a list of all the involved domains and all their public keys to the browser, and the latter uses the witnessed key as well as the list as the point of reference. Then, in subsequent connections, the browser verifies *(i)* that the public key which the server presents is contained in the list which was received during initialization, and *(ii)* that the server agrees on the legitimacy of the public key that was originally witnessed by the browser during initialization. Notice how this differs from pinning, which operates under the assumption that the initial connection is trusted, and thus does not seek to verify the legitimacy of the initial connection, and consequently of the received pins, upon subsequent connections.

The above technique is indeed useful when considering MITM+certificate attacks and can be used to implement the server invariance protocol in SISCA. Nevertheless, in the following sections we present an alternative approach that does not leverage server public keys, and aims to mitigate MITM+key attacks, as well. We note that the security analysis as well as most of the design patterns that are discussed in the approach that follows (e.g., how to prevent downgrade attacks and allow for partial support and exceptions – Section 3.2.6, how to secure resource caching – Section 3.2.7, etc) would similarly apply to the previously sketched technique, too.

Our High-Level Approach. In SISCA we choose to implement server invariance as a simple challenge/response protocol. In the initialization phase (first connection) the browser sets up a fresh challenge/response pair (which acts as the point of reference) with the server. Then, in the verification phase (subsequent connections) the browser challenges the server to verify server invariance, i.e., that it is the same entity with which the browser executed the initialization.

SISCA has to be executed before any HTTP traffic influenced by the attacker is processed by the browser or the server. We choose to implement the protocol at the application layer, over established TLS sessions via an HTTP header, named `X-Server-Inv`, and transmitted together with the *first* HTTP request/response pair over a particular TLS connection. For the protocol to be secure, on the client side this header is controlled solely by the browser. It cannot be created or accessed programmatically via scripts (similar to cookie-related headers [3]).

Alternatively, we could implement the server invariance protocol in SISCA as a TLS extension, i.e., at the transport layer. We deem the application layer more appropriate, since server invariance encompasses semantics that are naturally offered by the application layer, such as cross-origin interaction and content inclusion.

Figure 7 depicts a simple example of how a protocol based on our approach can look like. In this example

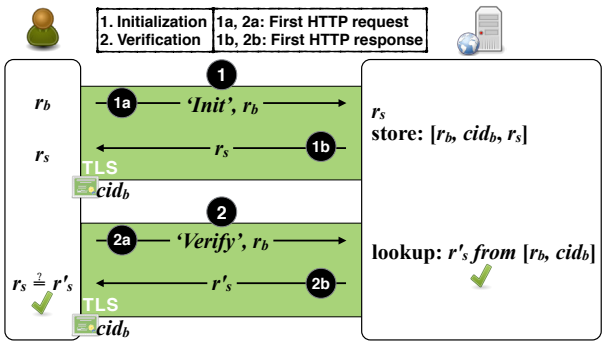


Figure 7: An example challenge/response-based server invariance protocol requiring per-client server state.

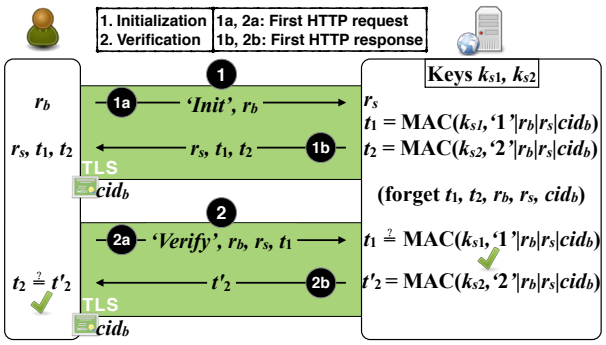


Figure 8: Basic SISCA protocol.

protocol, during the initialization phase the browser and server generate random numbers r_b and r_s, which they both store (the server also stores the browser's Channel ID cid_b). The browser subsequently uses r_b as a challenge during the verification phase, expecting the response r_s by the server. The latter looks up r_s by using r_b and cid_b. For the shake of brevity, we do not analyze this example, but we make the following important remarks.

First, this example requires the server to store per-client state. This may be undesirable and it also makes it harder for multiple servers belonging to the same entity to share the common state which is needed in order to be able to correctly execute the protocol. For this reason, SISCA uses symmetric cryptography (MAC), in order to securely offload the state to the clients.

Second, during the verification phase, the server should process the incoming HTTP request, only if the lookup succeeds. If it fails, it means that the attacker intercepted the first connection (initialization phase) and that the incoming request may be malicious. We explain this concept further in the analysis of the SISCA protocol. We note that due to this fact, SISCA uses a second MAC tag in order to enable the server perform this check.

3.2.5 Basic Protocol

We now describe the server invariance protocol of SISCA in detail. We follow a structural approach, meaning that we start with a basic version of our protocol, described in this section. Then, in subsequent sections, we incrementally add features.

Figure 8 illustrates the protocol, assuming no attack. Prior to the protocol execution, the server, `www.example.com`, generates two keys k_{s1} and k_{s2}, called *SISCA keys*. The same SISCA keys are used for all protocol executions (i.e., not for a specific client) and are never disclosed to other parties. Moreover, recall that the server and client deploy Channel ID-based authentication. Each TLS connection will therefore have a Channel ID cid_b, that is created by the user's browser. As already mentioned, the protocol consists of two phases.

Initialization. The initilization phase occurs once the browser establishes a TLS connection to `www.example.com`, for the *first time* in a browsing session (upper connection in Figure 8). The browser picks a random number r_b. It then sends $\langle$'*Init*',$r_b\rangle$ to the server ('*Init*' is a string constant), within the *first* HTTP request[3] over that connection. Upon receiving this message, the server chooses a random number r_s and computes the following message authentication tags:

$$t_1 = \text{MAC}(k_{s1}, \text{'1'}|r_b|r_s|cid_b) \quad (1)$$

$$t_2 = \text{MAC}(k_{s2}, \text{'2'}|r_b|r_s|cid_b) \quad (2)$$

where '1' and '2' are strings constants. Notice that the server binds the computed tags to the browser's Channel ID cid_b. r_b, r_s and the MAC tags will be used in subsequent TLS connections to verify server invariance.

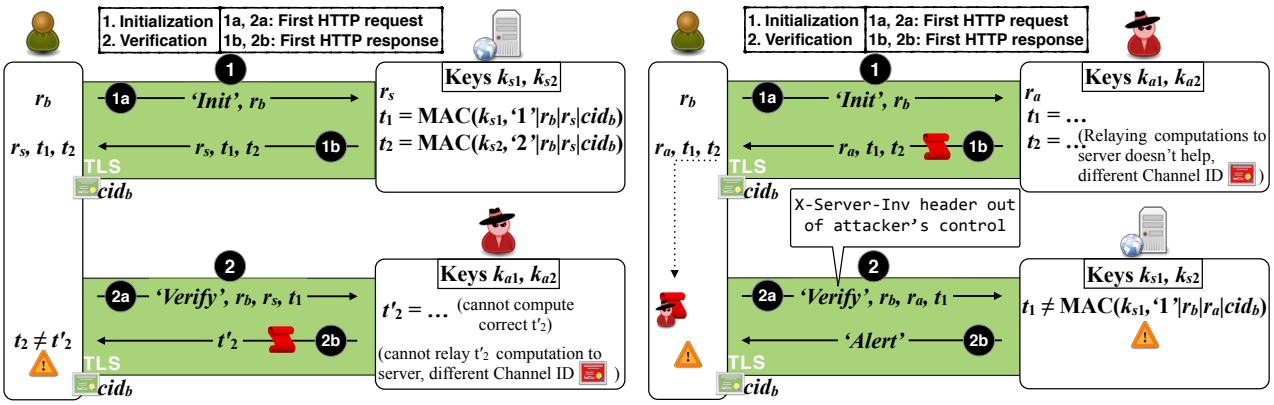
Finally, the server sends $\langle r_s,t_1,t_2\rangle$ to the browser within its first HTTP response. The browser stores $\langle r_b,r_s,t_1,t_2\rangle$, while the server does not store any client-specific information. At this point, the initialization phase is complete. Subsequent HTTP requests and responses over that particular TLS connection do *not* include an `X-Server-Inv` header.

Verification. The verification phase takes place upon every subsequent TLS connection to `www.example.com`, which occurs within the same browsing session (lower connection in Figure 8). Like in the first phase, the protocol messages are exchanged within the *first* HTTP request/response pair. The browser sends $\langle$'*Verify*',$r_b,r_s,t_1\rangle$ to the server, as part of the first request. After receiving the request, and *before* processing it, the server first checks if

$$t_1 \stackrel{?}{=} \text{MAC}(k_{s1}, \text{'1'}|r_b|r_s|cid_b). \quad (3)$$

Here, cid_b corresponds to the Channel ID of the TLS session within which the protocol is currently being exe-

[3]Note that this is a request that browser would anyway submit, i.e., required for loading the web page. It is not an extra request.



(a) Attacker intercepts the verification phase.

(b) Attacker intercepts the initialization phase.

Figure 9: Resilience of SISCA to MITM-SITB (conventional MITM is prevented by Channel-ID based authentication).

cuted, which, if under attack, might differ from the Channel ID that was used in the initialization phase. If the check passes, the server computes

$$t'_2 = \text{MAC}(k_{s2}, \text{'2'}|r_b|r_s|cid_b), \quad (4)$$

processes the received request, and passes $\langle t'_2 \rangle$ within the HTTP response to the browser. Finally, the browser checks if $t'_2 \stackrel{?}{=} t_2$ and if it succeeds, it means that server invariance holds for this TLS connection.

Analysis When Under Attack. Figure 9 illustrates how the protocol prevents MITM attacks. Recall that, due to the usage of Channel ID-based authentication, the attacker cannot perform the conventional attack (Figures 1, 2) – the attacker's TLS sessions will have a different Channel ID than the client's and will thus be rejected. Instead, she has to execute a MITM-SITB attack.

In Figure 9 we illustrate two possible attack scenarios (based on the discussion of Section 3.2.3) and we show why the attacker fails in both. In Figure 9a the attacker intercepts the verification phase of SISCA. Since the attacker didn't participate in the initialization phase of the protocol, she does not know the correct MAC response t_2 to the client's challenge. Moreover, since she does not have access to k_{s2}, she cannot calculate the correct t_2 either (Eq. (4)). As a result, the user's browser rejects the attacker's response and terminates the session, notifying the user (no user decision is required). Even if the attacker pushes a malicious script in her response, it will not get a chance of being executed.

In the second scenario, depicted in Figure 9b, the attacker intercepts the first TLS connection to `www.example.com`. She thus executes the initialization phase with the browser and injects her script, which is executed within the web origin of `www.example.com`. To successfully complete her attack, the attacker needs to let a subsequent TLS connection reach the legitimate server, and access the user's account via that connection.

After the browser establishes a connection with the legitimate server, the two of them execute the verification phase, as part of the first HTTP request/response pair. The server, before processing the HTTP request (which might as well be malicious), checks whether Condition (3) is true. Since the attacker does not have access to key k_{s1}, she could not have computed the correct t_1 (Eq. (1)). Thus, during the initialization phase, she sends a t_1 value to the browser that is not the correct one. Consequently, Condition (3) will not be satisfied. In this case the server does not process the request, and instead notifies the browser by sending an empty HTTP response containing $\langle$'*Alert*'$\rangle$ in the `X-Server-Inv` header. This indicates violation of the server invariance and the browser aborts the session.

We remark that in the second scenario, it is the legitimate server that checks server invariance, detects the ongoing MITM attack and notifies the browser. This is important in order to prevent even a single malicious request from being accepted and processed by the server.

We conclude our analysis, with a few remarks that are relevant for both of the scenarios described above. First, note that the attacker cannot relay any of the necessary MAC computations to the legitimate server. In other words, she cannot manipulate the server to compute for her the values needed for cheating in the protocol. This is because the server binds all its computations to the channel ID of the client with whom it communicates (the attacker's channel ID will be different from the user's).

Second, note that the protocol is secure so long as the attacker cannot "open" already established TLS connections between the browser and the legitimate server (i.e., connections that she chose not to intercept). If she could do that, she would be able to extract the correct values of both t_1 and t_2 and successfully cheat. Recall that, the MITM+key attacker holds the private key of the legitimate server. Therefore, in order to prevent such an at-

tacker from eavesdropping on already established TLS connections, it is essential that these connections have TLS forward secrecy enabled.

Third, when considering MITM+key attacks it is reasonable to assume that the attacker can also extract the SISCA keys, similar to the private key of the server. As stated in the assumptions (Section 3.2.2) and explained in Section 3.2.8, SISCA keys, unlike the private key, can be frequently rotated. SISCA can thus resist MITM+key attacks, assuming no persistent server compromise.

Finally, the attacker can choose not to reply at all, when executing SISCA with the user. This essentially leads to a Denial of Service (DoS) attack. However, such attacks can already be achieved even by attackers less powerful that those considered here. That is, attackers that cannot perform TLS MITM attacks, but can block network traffic between the browser and the server.

Different Origins. The SISCA protocol execution is guided by the same-origin policy [5]. In particular, SISCA is executed independently, i.e., different *protocol instances*, when loading web pages and documents that belong to different origins. For example, assume that the browser navigates to `www.example.com` for the first time in the current browsing session. Then, a new instance of SISCA will be created for this origin and its initialization phase will be executed on the first TLS connection. If the browser further navigates to pages belonging to `www.example.com`, and this triggers the creation of new TLS connections by the browser, then for those connections the browser will execute the verification phase of the previously created SISCA instance corresponding to `www.example.com` (same origin). When the browser navigates to another website (different origin), say `www.another.com`, then a new instance of SISCA will be created and used for the loading of documents from that origin (assuming that this is the first visit to `www.another.com` in that browsing session). Also any HTTP redirections during navigation that lead to different origins will cause the corresponding SISCA instances for those origins to be created and used.

3.2.6 Cross-Origin Communication

In the previous section we assumed that accessing the web pages of `www.example.com` involves communication only with that domain, i.e., web origin. However, this is not a realistic scenario in today's web applications. Many websites perform cross-origin requests, e.g., to load resources. SISCA can accommodate for such scenarios so long as all the involved domains belong to, and are administered by the *same entity*, such that the required SISCA keys, k_{s1} and k_{s2}, can be shared across all relevant servers.

Therefore, for cross-origin communication the browser uses the SISCA instance corresponding to the initiating origin. For example, assume that a page loaded from `www.example.com` performs a cross-origin request to `static.example.com`. The browser will create a TLS connection to `static.example.com` and will execute the verification phase of the SISCA instance that corresponds to `www.example.com`. Any potential HTTP redirections will also use the SISCA instance of the initiating origin, `www.example.com`.

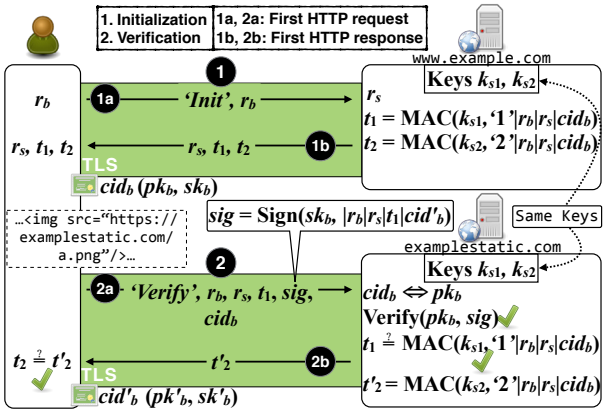


Figure 10: SISCA adapted for cross-origin communication (the origins share the same SISCA keys), when the browser uses a different Channel ID for each origin. Here, `www.example.com` performs a cross-origin request to `examplestatic.com`.

Different Channel IDs. The basic protocol we described in Section 3.2.5 also works in the cross-origin communication scenario, provided that the Channel ID used by the browser is the *same*. The Channel ID specification draft already recommends using the same Channel ID for a domain and its subdomains [4, 15] (to account for cookies that have the `Domain` attribute set). For example, the browser should use the same Channel ID for `www.example.com` and `static.example.com`. Nevertheless, for privacy reasons, the specification recommends using different Channel IDs for unrelated domains. In such cases, SISCA has to account for using different Channel IDs across domains, when cross-origin communication takes place.

Figure 10 depicts how the protocol works in such a scenario. The browser navigates to `www.example.com`, and starts a new SISCA instance for that origin. The browser uses Channel ID cid_b (with public key pk_b, and private key sk_b). At some later point in time, the page loaded from `www.example.com` performs a cross-origin request to `examplestatic.com`, which is controlled by the same entity. Nevertheless, since it corresponds to a different domain (i.e., not a subdomain), the browser uses a different Channel ID, say cid'_b (with pk'_b, sk'_b being the corresponding public/private key pair). In this case, although the initialization phase of SISCA was ex-

ecuted using cid_b, the verification phase will have to be executed over a TLS connection with Channel ID cid'_b.

As Figure 10 shows, the browser needs to tell the server (`examplestatic.com`) to use cid_b instead of cid'_b, but do so in a secure way. To achieve this, the browser endorses cid'_b, by signing it with sk_b, and thus proving to the server that it owns the private keys of both Channel IDs cid_b and cid'_b. The browser extends the '$Verify$' message by appending cid_b and a signature over cid'_b (i.e., the Channel ID of that TLS connection) and the rest of the message parameters using sk_b. The server, before processing the request, verifies the signature on cid'_b using the supplied cid_b (i.e., pk_b). If it passes, then the server uses cid_b for the subsequent steps of the verification phase, which remain unchanged.

Overlapping Cross-Origin Access. Browsers typically send multiple HTTP requests over the same network connection (persistent connections [23]). Due to the existence of cross-origin communication, a TLS connection to a particular domain, say `static.example.com`, can be used by the browser to transmit cross-origin requests to `static.example.com` made by different initiating origins. For example, the browser uses the same TLS connection to `static.example.com`, to transmit, first, a request originating from a document belonging to `www.example.com` and then, a request originating from a document belonging to `shop.example.com` (we still assume that all three domains belong to the same entity). In this case, the TLS connection to `static.example.com` has to be verified using SISCA for both initiating domains, independently.

In the above scenario, the browser executes the verification phase with the SISCA instance of `www.example.com`, upon establishing the TLS connection to `static.example.com` and sending the first HTTP request, originating from `www.example.com`. Subsequently, when the browser wants to reuse the same connection to send a cross-origin request from `shop.example.com` to `static.example.com`, it once again executes the verification phase, only this time with the SISCA instance of `shop.example.com`. This takes place upon transmitting the first HTTP request, which originates from `shop.example.com`.

Origin Change. A web page is allowed to change its own origin (effective origin) to a suffix of its domain, by programmatically setting the value of `document.domain` [40]. This allows two pages belonging to different subdomains, but presumably to the same entity, to set their origin to a common value and enable interaction between them[4]. For example, a page from `www.example.com` and a page from `shop.example.com` can both set their origin to `example.com`. In such a case, the attacker can attack the user account at `shop.example.com`, by intercepting the first connection to `www.example.com` (or any other `example.com` subdomain), or vice versa.

To prevent such an attack, the browser has to verify that server invariance holds across each pair of origins that change their effective origin to a common value, before allowing any interaction between them. Each origin has its own SISCA instance established, and we must ensure that both SISCA instances were initialized with the same remote entity. This can be achieved by running the verification phase of both instances over the same TLS connection (established to either origin). The browser can reuse an already established and verified connection with one origin, and just verify the connection with the SISCA instance of the other origin. If no such connection exists at that time, then the browser can create a new one to either origin and execute the verification phase of both SISCA instances. If there is no actual HTTP request to be sent at that time, the browser can make use of an HTTP `OPTIONS` request.

Partial Support and Downgrade Attacks. SISCA must be incrementally deployable, which means that it must maintain compatibility with legacy web servers, without compromising the security of the SISCA-enabled servers. Moreover, websites must be able to opt for partial support. As an example, a domain implements SISCA but still needs to perform cross-origin requests to another domain, called *incompatible*, that either does not support SISCA, or supports it but belongs to a 3rd party, i.e., it has different SISCA keys (we discuss on the security of such design choices at the end of this section).

The above can be achieved by allowing *exceptions*. If a particular domain does not support SISCA (including legacy servers that are not aware of SISCA at all), then it can simply ignore the `X-Server-Inv` header, sent during the initialization phase, and reply without including any SISCA-related information. This will be received by the browser as an *exception claim*. Moreover, if a domain supports SISCA but performs cross-origin communication with one or more incompatible domains, then it can append an *exception list* in its response, during the initialization phase, designating the incompatible domains.

However, we note that if the attacker intercepts the initialization phase of the protocol, then she could perform a *protocol downgrade attack*, by providing false exception claims or exception lists in her response.

To prevent downgrade attacks, the browser should verify any exception that was received during the initialization phase, upon every subsequent connection. If the attacker intercepted the initialization phase and replied with fake exception claims, then if any of the subsequent connections reaches the legitimate server, the browser, with the help of the legitimate server, would detect the

[4] Both pages have to explicitly set `document.domain`.

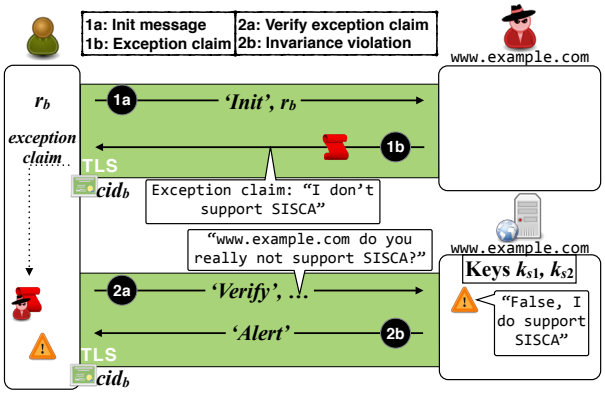


Figure 11: Preventing downgrade attacks (same-origin case).

attack. This scenario is illustrated in Figure 11.

Regarding cross-origin communication, in order to help SISCA-enabled legitimate servers detect fake exception lists previously received by the browser, SISCA protocol messages should include (in the `X-Server-Inv` header) the origin associated with the SISCA instance. Suppose for example, that the browser executes the initialization phase with `www.example.com` which supports SISCA (executes the protocol normally), but also includes an exception list stating that it performs cross-origin requests to `shop.example.com` which does not support SISCA. Whenever the browser connects to `shop.example.com` to perform a cross-origin request from `www.example.com`, the browser includes the origin of the SISCA instance (`www.example.com`) and asks `shop.example.com` whether it indeed does not support SISCA with respect to that origin. Assuming that the connection was not intercepted, `shop.example.com` can leverage the supplied origin information to decide whether the exception reported by the browser is valid. If not, then it should abort processing the request and notify the browser of the detected attack. Note that the above assumes that each SISCA-enabled server is aware of all the domains that is compatible to execute SISCA with (i.e., domains with which it shares the same SISCA keys), which is not difficult to implement.

3rd Party Content Inclusion. As mentioned above, a domain, say `www.example.com`, implementing SISCA can still perform cross-origin requests to incompatible 3rd party domains as long as it designates those domains as exceptions for the protocol. This of course means that TLS connections to those domains will not be protected by SISCA, and could be MITM-ed by the attacker to perform a user impersonation attack on `www.example.com`. This can be indeed the case if `www.example.com` includes active content [39] (in particular, JavaScript and CSS) from those domains. Embedding JavaScript from 3rd party sites is generally not recommended, and usually there are ways of avoiding it [41]. Furthermore, depending on the use case, it may be possible to use *iframes* to isolate active 3rd party content, instead of directly embedding it within the target origin, in order to mitigate the risk (the `sandbox` attribute can help even further).

The embedding of passive content only, such as images, does not give the attacker the ability to execute her code within the target origin. Hence, with respect to preventing user impersonation, such embeddings are safe and do not undermine the security offered by SISCA.

3.2.7 Resource Caching

Caching of static resources, such as scripts and images, helps reduce web page loading times as well as server resource consumption. However, the way caching is currently implemented [23, 25] can give a MITM attacker the opportunity to subvert SISCA.

In brief, during one browsing session, the attacker intercepts all TLS connections and ensures that a legitimate, yet maliciously modified script that is required by the target web server is cached by the browser. Then, during a second browsing session, the attacker lets all connections pass through. When the legitimate web page asks for the inclusion of the aforementioned script, the browser will load it from cache, essentially enabling the execution of the attacker's malicious code. The attacker will thus be able to access the target web server.

To prevent the above attack, we need to change the way caching is performed for active content that would enable this attack (JavaScript and CSS files). We need to make sure that the browser always communicates with the server in order to verify that the cached version is the most recent and also the correct one (i.e., not maliciously modified). Thus, caching of such files should be performed only using Entity Tags (ETags) [23], but in a more rigorous way than specified in the current HTTP specification. In particular, if a web server wishes to instruct a browser to cache a JavaScript or CSS file, the server should use an `ETag` header which always contains a cryptographic hash of the file. The browser, before using, and caching the file should *verify* that the supplied hash is correct. Subsequently, before the browser uses the cached version of the file, it first verifies that the local version matches the version of the server (using the `If-None-Match` header, as currently done).

3.2.8 Key Rotation

In SISCA, the server has a pair of secret keys, k_{s1} and k_{s2}. To resist key compromise (i.e., MITM+key attackers), these keys, unlike the server's private key, can be easily rotated. This is because the SISCA keys need not undergo any certification process, and can thus be rotated frequently, e.g., weekly, daily, or even hourly. The more frequent the rotation the smaller the attacker's window of

opportunity to successfully mount MITM attacks.

The key transition, of course, has to be performed such that it does not break the execution of active browser SISCA instances that rely on the previous keys. At a high level, one way of achieving this, is to have the server keep previous keys for a certain period of time (i.e., allow partial overlap of keys). This can enable browsers with active SISCA instances that rely on the previous keys to securely transition to new protocol parameters, i.e., t_1 and t_2, computed using the new server SISCA keys.

For domains served by a single machine, this is only a matter of implementing the corresponding functionality in the web server software (e.g., Apache). For multiple domains controlled by the same entity and served by multiple machines, located in the same data center or even in different data centers across the world, arguably more effort is required in order to distribute the ever-changing keys and keep the machines in sync. Nevertheless, a similar mechanism is needed for enabling TLS forward secrecy while supporting TLS session tickets [34]. According to Twitter's official blog [30], Twitter engineers have implemented such a key distribution mechanism.

3.2.9 SISCA Benefits and Drawbacks

SISCA offers the following advantages regarding MITM+certificate attack prevention. Compared to multi-path probing solutions, SISCA does not rely on any third party infrastructure, trusted or not. Since SISCA is built on top of Channel ID-based authentication, it has to assume that no MITM attack takes place during user enrollment. Nevertheless, after this step, no "blind" trust is required when the user uses a new or clean browser, contrary to pinning solutions (except preloaded pins), as discussed in Section 3.2.4. Moreover, in SISCA no user decision is necessary whenever server invariance violation is detected. This can occur either due to an attack, or due to an internal server fault, thus the browser can abort (possibly after retrying) the session. SISCA is scalable since it can be deployed incrementally by web providers (assuming browser support). Finally, SISCA resists MITM+key attacks, assuming that the attacker does not persistently compromise the server.

The main disadvantage of SISCA is that it only protects against MITM attackers whose goal is to impersonate the user to the server. This is arguably the most common and impactful attacker goal. SISCA does not protect against attackers whose objective is to provide fake content to the user. In such cases the attacker can simply intercept *all* connections and interact with the user by serving her own, fake content. In contrast, the techniques that focus on ensuring the correctness of server authentication (Section 3.1) can protect against such attacks (MITM+certificate attackers). As a result, a recommended strategy would be to use SISCA in conjunction with any of these techniques. Finally, SISCA requires coordination between an entity's different domains, in the sense they must have access to the same SISCA keys. This is needed for securing cross-origin communication and, depending on the scale of the entity, can be challenging from an engineering perspective to set up.

3.2.10 Interaction With Other Web Technologies

SPDY. SPDY [6] multiplexes concurrent HTTP requests over the same TLS connection to improve network performance. In order for SISCA to be compatible with the general SPDY functionality, the browser must ensure that before the SISCA protocol is completed successfully (i.e., the first request/response pair is exchanged), no further requests are sent through the SPDY connection.

Furthermore, SPDY IP Pooling allows, under certain circumstances, HTTP sessions from the same browser to different domains (web origins) to be multiplexed over the same connection. Version 3 of SPDY is compatible with Channel IDs (recall that different Channel IDs may need to be used for different origins, but now there is only one TLS connection). SISCA is compatible with IP Pooling, as long as the browser manages the multiplexed HTTP sessions independently, with respect to the execution of the SISCA protocol.

WebSocket. SISCA is compatible with the WebSocket protocol [21], when the latter is executed over TLS. This, of course assumes that *(i)* Channel IDs are used for the WebSocket TLS connections, *(ii)* the SISCA protocol is executed during the WebSocket handshake (i.e., first request/response pair), and *(iii)* JavaScript is not be able to manipulate the `X-Server-Inv` header.

Web Storage. Web Storage [27] is an HTML5 feature that allows a web application to store data locally in the browser. SISCA can protect `code.sessionStorage` (temporary storage), but does not prevent a MITM attacker from accessing information stored in `window.localStorage` (permanent storage), so no sensitive information should be stored there.

Offline Web Applications. HTML5 offers Offline Web Applications [26] which allow a website to create an offline version, stored locally in the browser. As with regular file caching (see Section 3.2.7), this feature can be leveraged by the attacker to bypass SISCA. Making this feature secure requires the introduction of design concepts similar to what we proposed for regular caching.

Other Client-Side Technologies. The attacker might attempt to leverage various active client-side technologies besides JavaScript, such as Flash, Java and Silverlight. Such technologies allow the attacker to create direct TLS connections to the legitimate server. Some of the APIs offered by those technologies also allow the attacker to forge and arbitrarily manipulate HTTP headers, including cookie-related headers or the `X-Server-Inv` header.

However, provided that Channel IDs and SISCA are not integrated with these technologies[5], the attacker will not be able to impersonate the user and compromise his account on the legitimate server.

3.3 Prototype SISCA Implementation

We created a proof of concept implementation of the basic SISCA protocol, with additional support for cross-origin communication, provided that the same Channel ID is used. On the server side we use Apache 2.4.7 with OpenSSL 1.0.1f, patched for Channel ID support. SISCA is implemented as an Apache module and consists of 313 lines of C code. On the client side we implement SISCA by modifying the source code of Chromium 35.0.1849.0 (252194) and the WebKit (Blink) engine. We make a total of 319 line modifications (insertions/deletions) in existing files and we add 6 new files consisting of 418 lines of C++ code.

We use Base64 encoding for binary data transmission. When using 128-bit random values (r_b and r_s) and HMAC-SHA256 (i.e., 256-bit tags, t_1 and t_2), the client's lengthiest message is 114 bytes long, plus the origin of the SISCA instance that has to be sent as well. The server's lengthiest message is 132 bytes long.

We finally verified that our implementation successfully blocks our proof of concept MITM-SITB attack.

Performance Evaluation. To assess the performance overhead imposed by SISCA (the server invariance part, not the overhead due to Channel IDs), we measured the latency of HTTP request/response roundtrips, with SISCA enabled and disabled. For the measurements we used a 4KB HTML page, as well as an 84KB jQuery compressed file, retrieved over a domain that we set up as being "cookieless". Chromium ran on a Macbook Pro laptop (2.3GHz CPU, 8GB RAM) and Apache ran on a typical server machine (six core Intel Xeon 2.53GHz, 12GB RAM), connected through the campus network.

We found that the overhead of the basic SISCA protocol is negligible, as no increase in latency was measured (averaged over 300 repetitions). Moreover, the HTTP request to the cookieless domain was able to fit in a single outgoing packet (a typically desired objective).

Regarding cross-origin communication over different Channel IDs (see Section 3.2.6), approximately 180 bytes are further added to the request (one ECDSA public key and signature in Base64 encoding), which can still fit in a single packet (for cookieless domain requests). Furthermore, the server has to perform one ECDSA signature verification. This overhead could be minimized, if the browser used the same Channel ID, not only for subdomains of the same domain, but also for domains belonging to the same entity. Although we do not elaborate on this idea here, this could be heuristically determined by the browser, based on which domains are involved in the execution of the same SISCA instance.

Finally, recall that a SISCA instance is executed only once per TLS connection and not on every HTTP request/response.

[5]This, for example, means that a TLS connection created by such an API will have to create and use its own Channel IDs, and that the browser will not execute SISCA over those connections.

4 Related Work

A significant amount of research in the past years surrounds the security of the TLS protocol, in the context of web applications (i.e., HTTPS), as well as web server and client authentication. A comprehensive overview is provided in [10], which, among others, surveys existing primitives that try to enhance the CA trust model in order to more effectively address MITM attacks.

The use of server impersonation for the compromise of the user's account by serving the attacker's script to the victim's browser was first introduced in [32]. In this attack, called *dynamic pharming*, the attacker exploits DNS rebinding vulnerabilities in browsers, by dynamically manipulating DNS records for the target server, in order to force the user's browser to connect either to the attacker (to inject her script) or to the legitimate server.

MITM-SITB is therefore very similar to dynamic pharming in that it leverages server impersonation to serve the script to the victim's browser. Dynamic pharming focuses on the attacker's ability to control the client's network traffic via DNS attacks, while in this paper we do not make such assumptions. Instead, MITM-SITB can leverage any form of MITM where the attacker controls the communication to the client (e.g., an attacker sitting on a backbone) and relies only on the behavior of the browser to re-establish a connection (with the legitimate server) once the attacker closes the connection within which she injected her script to the browser. Dynamic pharming can equally be used to successfully attack Channel ID-based solutions. Recently, the act of leveraging script injection via server impersonation against TLS client authentication was also discussed in [47].

We note that MITM-SITB (as well as dynamic pharming) differs from *Man-In-the-Browser* (MITB) [45]. The latter implies that the attacker is able to take full control of the browser by exploiting some vulnerability, or installing a malicious browser plugin. In MITM-SITB, the attacker runs normal JavaScript code within the target web origin and only within the boundaries established by the JavaScript execution environment. Therefore, no browser exploitation is required. Similarly, MITM-SITB is different from XSS [44]. In XSS the attacker is able to influence only parts of the served document (by exploiting a script injection vulnerability), while in MITM-SITB she is able to impersonate the server and thus influence the entire HTTP response sent to the browser.

SISCA does not prevent MITB or XSS and addressing these attacks is orthogonal to our work.

To prevent dynamic pharming, the locked same-origin policy (SOP) was proposed [32]. Weak locked SOP considers attackers with invalid certificates, while strong locked SOP also defends against attackers with valid, mis-issued certificates. Strong locked SOP refines the concept of origin by including the public key of the server and can also accommodate for multiple server keys. Strong locked SOP isolates web objects coming from connections with not endorsed server public keys in a separate security context (i.e., different origin). Strong locked SOP per se does not prevent a MITM attacker from mounting a conventional MITM attack in order to impersonate the user. A strong client authentication solution should be used in conjunction, as with SISCA.

Locked SOP does not resist MITM+key attacks, as SISCA does. Moreover, locked SOP is not able to secure cross-origin active content inclusion. The risks involved when a web page imports active content, such as JavaScript, that can be intercepted and modified by an attacker are discussed in [31]. SISCA can secure cross-origin inclusions as long as the involved domains belong to the same entity and thus share the same SISCA keys.

The current Channel ID specification [4] was recently found to be vulnerable to *triple handshake attacks* [7], which affect TLS client authentication in general. The mitigation proposed in [7] has already been implemented in the version of Chromium that we used in this work. SISCA assumes that Channel IDs work as expected, so eliminating triple handshake attacks is essential for its security. However, we note that addressing triple handshake attacks does not prevent MITM-SITB attacks.

Recent work has proposed leveraging Channel ID-based authentication to strengthen federated login [16] and Cloud authorization credentials [8], against MITM attacks and credential theft in general. However, such proposals fail to address MITM attacks, as they are susceptible to MITM-SITB, unless augmented with server invariance, as we propose in this paper with SISCA.

Server invariance is based on *sender invariance* which was formally defined in [17]. SISCA is inspired by this notion, assuming that the server's authenticity cannot be established via server certificate verification and instead trying to enforce the weaker property of invariance.

5 Conclusion

In this paper we discussed the requirements to effectively preventing TLS MITM attacks in the context of web applications, when the attacker's goal is to impersonate the user to the legitimate server and gain access to the user's account and data. Striving to defeat this type of attack is essential, especially given the recent revelations about government agencies (e.g., the NSA) mounting such attacks in order to perform mass surveillance against users of major internet services [18, 48].

We showed that strong client authentication alone, such as the recently proposed Channel ID-based authentication, cannot prevent such attacks. Instead, strong client authentication needs to be complemented with the concept of server invariance, which is a weaker and easier to enforce property than server authentication. Our solution, SISCA, shows that server invariance can be implemented with minimal additional cost on top of the proposed Channel ID-based approaches, and can be deployed incrementally, thus making it a scalable solution. Given its security benefits, we believe that SISCA can act as an additional, strong protection layer in conjunction with existing proposals that focus on amending today's server authentication issues, towards the effective prevention of TLS MITM attacks.

Acknowledgements

We thank our shepherd Dan Wallach, the anonymous reviewers, as well as Kari Kostiainen, Arnis Parsovs, Hubert Ritzdorf, Mario Strasser and Der-Yeuan Yu for their valuable feedback and insights. Some of the icons that are used in this work were taken and adapted from `opensecurityarchitecture.org`.

References

[1] The Heartbleed Bug. `http://heartbleed.com/`.

[2] ADKINS, H. An update on attempted man-in-the-middle attacks. `http://googleonlinesecurity.blogspot.ch/2011/08/update-on-attempted-man-in-middle.html`.

[3] AUBOURG, J., SONG, J., STEEN, H. R. M., AND VAN KESTEREN, A. XMLHttpRequest (W3C Working Draft). `http://www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/`.

[4] BALFANZ, D., AND HAMILTON, R. Transport Layer Security (TLS) Channel IDs, v01 (IETF Internet-Draf). `http://tools.ietf.org/html/draft-balfanz-tls-channelid-01`, 2013.

[5] BARTH, A. The web origin concept (RFC 6454). `http://tools.ietf.org/html/rfc6454`, 2011.

[6] BELSHE, M., AND PEON, R. SPDY protocol (IETF Internet-Draf). `http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00`, 2012.

[7] BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., PIRONTI, A., AND STRUB, P.-Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE SP (Oakland), 2014*.

[8] BIRGISSON, A., POLITZ, J. G., ERLINGSSON, U., TALY, A., VRABLE, M., AND LENTCZNER, M. Macaroons: Cookies with contextual caveats for decentralized authorization in the Cloud. In *NDSS, 2014*.

[9] BRIGHT, P. Independent Iranian Hacker Claims Responsibility for Comodo Hack. `http://www.wired.com/2011/03/comodo_hack/`.

[10] CLARK, J., AND VAN OORSCHOT, P. C. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE SP (Oakland), 2013*.

[11] COATES, M. Revoking trust in two TurkTrust certificates. https://blog.mozilla.org/security/2013/01/03/revoking-trust-in-two-turktrust-certficates/.

[12] CZESKIS, A., AND BALFANZ, D. Protected login. In *USEC, 2012*.

[13] CZESKIS, A., DIETZ, M., KOHNO, T., WALLACH, D., AND BALFANZ, D. Strengthening user authentication through opportunistic cryptographic identity assertions. In *CCS, 2012*.

[14] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) protocol, version 1.2 (RFC 5246). http://tools.ietf.org/html/rfc5246, 2008.

[15] DIETZ, M., CZESKIS, A., BALFANZ, D., AND WALLACH, D. S. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX Security, 2012*.

[16] DIETZ, M., AND WALLACH, D. S. Hardening Persona – Improving federated web login. In *NDSS, 2014*.

[17] DRIELSMA, P. H., MÖDERSHEIM, S., VIGANÒ, L., AND BASIN, D. Formalizing and analyzing sender invariance. In *FAST, 2006*.

[18] ECKERSLEY, P. A Syrian MITM attack against Facebook. https://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook.

[19] ECKERSLEY, P. The Sovereign Keys project. https://www.eff.org/sovereign-keys.

[20] EVANS, C., PALMER, C., AND SLEEVI, R. Public key pinning extension for HTTP (IETF Internet-Draf). http://tools.ietf.org/html/draft-ietf-websec-key-pinning-09, 2013.

[21] FETTE, I., AND MELNIKOV, A. The WebSocket protocol (RFC 6455). http://tools.ietf.org/html/rfc6455, 2011.

[22] FIDO ALLIANCE. Universal 2nd Factor (U2F) overview, Version 1.0 (review draft). http://fidoalliance.org/specs/fido-u2f-overview-v1.0-rd-20140209.pdf.

[23] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616). http://tools.ietf.org/html/rfc2616, 1999.

[24] GOOGLE DEVELOPERS. Minimize request overhead. https://developers.google.com/speed/docs/best-practices/request.

[25] GOOGLE DEVELOPERS. Optimize caching. https://developers.google.com/speed/docs/best-practices/caching.

[26] HICKSON, I. Offline web applications (HTML 5 working draft). http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html.

[27] HICKSON, I. Web storage (W3C Recommendation). http://www.w3.org/TR/webstorage/.

[28] HIGGINS, P. Pushing for perfect forward secrecy, an important web privacy protection. https://www.eff.org/deeplinks/2013/08/pushing-perfect-forward-secrecy-important-web-privacy-protection.

[29] HOFFMAN, P., AND SCHLYTER, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) protocol: TLSA (RFC 6698). http://tools.ietf.org/html/rfc6698, 2012.

[30] HOFFMAN-ANDREWS, J. Forward secrecy at Twitter. https://blog.twitter.com/2013/forward-secrecy-at-twitter-0.

[31] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *Web 2.0 Security and Privacy, 2008*.

[32] KARLOF, C., SHANKAR, U., TYGAR, J. D., AND WAGNER, D. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS, 2007*.

[33] KIM, T. H.-J., HUANG, L.-S., PERRIG, A., JACKSON, C., AND GLIGOR, V. Accountable Key Infrastructure: A proposal for a public-key validation infrastructure. In *WWW, 2013*.

[34] LANGLEY, A. How to botch TLS forward secrecy. https://www.imperialviolet.org/2013/06/27/botchingpfs.html.

[35] LANGLEY, A. Protecting data for the long term with forward secrecy. http://googleonlinesecurity.blogspot.ch/2011/11/protecting-data-for-long-term-with.html.

[36] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency (RFC 6992). http://tools.ietf.org/html/rfc6962, 2013.

[37] MARLINSPIKE, M. Convergence. http://convergence.io/.

[38] MARLINSPIKE, M., AND PERRIN, T. Trust Assertions for Certificate Keys (TACK) (IETF Internet-Draft). http://tack.io/draft.html, 2013.

[39] MOZILLA DEVELOPER NETWORK. Mixed content. https://developer.mozilla.org/en-US/docs/Security/MixedContent.

[40] MOZILLA DEVELOPER NETWORK. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript.

[41] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote Javascript inclusions. In *CCS, 2012*.

[42] OPPLIGER, R., HAUSER, R., AND BASIN, D. SSL/TLS session-aware user authentication - Or how to effectively thwart the man-in-the-middle. *Computer Communications 29*, 12 (2006), 2238–2246.

[43] OPPLIGER, R., HAUSER, R., AND BASIN, D. SSL/TLS session-aware user authentication revisited. *Computers & Security 27*, 3-4 (2008), 64–70.

[44] OWASP. Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[45] OWASP. Man-in-the-browser attack. https://www.owasp.org/index.php/Man-in-the-browser_attack.

[46] PAOLA, S. D., AND FEDON, G. Subverting Ajax. 23rd Chaos Communication Congress, 2006.

[47] PARSOVS, A. Practical issues with TLS client certificate authentication. In *NDSS, 2014*.

[48] SCHNEIER, B. New NSA leak shows MITM attacks against major Internet services. https://www.schneier.com/blog/archives/2013/09/new_nsa_leak_sh.html.

[49] SOGHOIAN, C., AND STAMM, S. Certified lies: Detecting and defeating government interception attacks against SSL. In *FC, 2011*.

[50] STERNE, B., AND BARTH, A. Content Security Policy 1.0 (W3C Candidate Recommendation). http://www.w3.org/TR/CSP/.

[51] SUNSHINE, J., EGELMAN, S., ALMUHIMEDI, H., ATRI, N., AND CRANOR, L. F. Crying wolf: An empirical study of SSL warning effectiveness. In *USENIX Security, 2009*.

[52] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX ATC, 2008*.

Scheduler-based Defenses against Cross-VM Side-channels

Venkatanathan Varadarajan
University of Wisconsin

Thomas Ristenpart
University of Wisconsin

Michael Swift
University of Wisconsin

Abstract

Public infrastructure-as-a-service clouds, such as Amazon EC2 and Microsoft Azure allow arbitrary clients to run virtual machines (VMs) on shared physical infrastructure. This practice of multi-tenancy brings economies of scale, but also introduces the threat of malicious VMs abusing the scheduling of shared resources. Recent works have shown how to mount cross-VM side-channel attacks to steal cryptographic secrets. The straightforward solution is *hard isolation* that dedicates hardware to each VM. However, this comes at the cost of reduced efficiency.

We investigate the principle of *soft isolation*: reduce the risk of sharing through better scheduling. With experimental measurements, we show that a *minimum run time (MRT) guarantee* for VM virtual CPUs that limits the frequency of preemptions can effectively prevent existing Prime+Probe cache-based side-channel attacks. Through experimental measurements, we find that the performance impact of MRT guarantees can be very low, particularly in multi-core settings. Finally, we integrate a simple per-core CPU state cleansing mechanism, a form of hard isolation, into Xen. It provides further protection against side-channel attacks at little cost when used in conjunction with an MRT guarantee.

1 Introduction

Public infrastructure-as-a-service (IaaS) clouds enable the increasingly realistic threat of malicious customers mounting side-channel attacks [35, 46]. An attacker obtains tenancy on the same physical server as a target, and then uses careful timing of shared hardware components to steal confidential data. Damaging attacks enable theft of cryptographic secrets by way of shared per-core CPU state such as L1 data and instruction caches [46], despite customers running within distinct virtual machines (VMs).

A general solution to prevent side-channel attacks is *hard isolation*: completely prevent sharing of particular sensitive resources. Such isolation can be obtained by avoiding multi-tenancy, new hardware that enforces cache isolation [42, 44], cache coloring [34, 36], or software systems such as StealthMem [22]. However, hard isolation reduces efficiency and raises costs because of stranded resources that are allocated to a virtual machine yet left unused.

Another approach has been to prevent attacks by adding noise to the cache. For example, in the Düppel system the guest operating system protects against CPU cache side-channels by making spurious memory requests to obfuscate cache usage [47]. This incurs overheads, and also requires users to identify the particular processes that should be protected.

A final approach has been to interfere with the ability to obtain accurate measurements of shared hardware by removing or obscuring time sources. This can be done by removing hardware timing sources [28], reducing the granularity of clocks exposed to guest VMs [41], allowing only deterministic computations [6], or using replication of VMs to normalize timing [26]. These solutions either have significant overheads, as in the last solution, or severely limit functionality for workloads that need accurate timing.

Taking a step back, we note that in addition to sharing resources and having access to fine-grained clocks, shared-core side-channel attacks also require the ability to measure the state of the cache *frequently*. For example, Zhang et al.'s cross-VM attack on ElGamal preempted the victim every 16 µs on average [46]. With less frequent interruptions, the attacker's view of how hardware state changes in response to a victim becomes obscured. Perhaps surprisingly, then, is the lack of any investigation of the relationship between CPU scheduling policies and side-channel efficacy. In particular, scheduling may enable what we call *soft isolation*: limiting the frequency of potentially dangerous cross-VM interactions. (We use

the adjective soft to indicate allowance of occasional failures, analogous to soft real-time scheduling.)

Contributions. We evaluate the ability of system software to mitigate cache-based side-channel attacks through scheduling. In particular, we focus on the type of mechanism that has schedulers ensure that CPU-bound workloads cannot be preempted before a minimum time quantum, even in the presence of higher priority or interactive workloads. We say that such a scheduler offers a *minimum run time (MRT) guarantee*. Xen version 4.2 features an MRT guarantee mechanism for the stated purpose of improving the performance of batch workloads in the presence of interactive workloads that thrash their cache footprint [11]. A similar mechanism also exists in the Linux CFS scheduler [29].

Cache-based side-channel attacks are an example of such highly interactive workloads that thrash the cache. One might therefore hypothesize that by reducing the frequency of preemptions via an MRT guarantee, one achieves a level of soft isolation suitable for mitigating, or even preventing, a broad class of shared-core side-channel attacks. We investigate this hypothesis, providing the first analysis of MRT guarantees as a defense against cache-based side-channel attacks. With detailed measurements of cache timing, we show that even an MRT below 1 ms can defend against existing attacks.

But an MRT guarantee can have negative affects as well: latency-sensitive workloads may be delayed for the minimum time quantum. To evaluate the performance impact of MRT guarantees, we provide extensive measurements with a corpus of latency-sensitive and batch workloads. We conclude that while worst-case latency can be hindered by large MRTs in some cases, in practice Xen's existing core load-balancing mechanisms mitigate the cost by separating CPU-hungry batch workloads from latency-sensitive interactive workloads. As just one example, memcached, when running alongside batch workloads, suffers only a 7% overhead on 95^{th}-percentile latency for a 5 ms MRT compared to no MRT. Median latency is not affected at all.

The existing MRT mechanism only protects CPU-hungry programs that do not yield the CPU or go idle. While we are aware of no side-channel attacks that exploit such victim workloads, we nevertheless investigate a simple and lightweight use of CPU *state cleansing* to protect programs that quickly yield the CPU by obfuscating predictive state. By implementing this in the hypervisor scheduler, we can exploit knowledge of when a cross-VM preemption occurs and the MRT has not been exceeded. This greatly mitigates the overheads of cleansing, attesting to a further value to soft-isolation style mechanisms. In our performance evaluation of this mechanism, we see only a 10–50 µs worse-case overhead on median latency due to cleansing while providing protection for all guest processes within a VM (and not just select ones, as was the case in Düppel). In contrast, other proposed defenses have similar (or worse) overhead but require new hardware, new guest operating systems, or restrict system functionality.

Outline. In the next section, we provide background on cache-based side-channel attacks and existing defense mechanisms. In Section 3 we describe the Xen hypervisor scheduling system, its MRT mechanism, and the principle of soft isolation. In Section 4 we measure the effectiveness of MRT as a defense. Section 5 shows the performance of Xen's MRT mechanism, and Section 6 describes combining MRT with cache cleansing.

2 Background and Motivation

Our work is motivated by the increasing importance of threats posed by side-channel attacks in multi-tenant clouds, in which VMs from multiple customers run on the same physical hardware. We focus on cache-based side-channels, which are dangerous because they can leak secret information such as encryption keys and have been demonstrated between virtual machines in a cloud environment [46].

Side-channel attacks. We can delineate side-channel attacks into three classes: time-, trace-, and access-driven. Time-driven attacks arise when an attacker can glean useful information via repeated observations of the (total) duration of a victim operation, such as the time to compute an encryption (e.g., [5, 7, 10, 16, 24]). Trace-driven attacks work by having an attacker continuously monitor a cryptographic operation, for example via electromagnetic emanations or power usage leaked to the attacker (e.g., [14, 23, 33].

We focus on access-driven side-channel attacks, in which the attacker is able to run a program on the same physical server as the victim. These abuse stateful components of the system shared between attacker and victim program, and have proved damaging in a wide variety of settings, including [3, 15, 31, 32, 35, 45]. In the cross-process setting, the attacker and victim are two separate processes running within the same operating system. In the cross-VM setting, the attacker and victim are two separate VMs running co-resident (or co-tenant) on the same server. The cross-VM setting is of particular concern for public IaaS clouds, where it has been shown that an attacker can obtain co-residence of a malicious VM on the same server as a target [35]

Zhang, Juels, Reiter, and Ristenpart (ZJRR) [46] demonstrated the first cross-VM attack with sufficient

granularity to extract ElGamal secret keys from the victim. They use a version of the classic Prime+Probe technique [31]: the attacker first *primes* the cache (instruction or data) by accessing a fixed set of addresses that fill the entire cache. He then yields the CPU, causing the hypervisor to run the victim, which begins to evict the attacker's data or instructions from various cache. As quickly as possible, the attacker preempts the victim, and then *probes* the cache by again accessing a set of addresses that cover the entire cache. By measuring the speed of each cache access, the attacker can determine which cache lines were displaced by the victim, and hence learn some information about which addresses the victim accessed.

The ZJRR attack builds off a long line of cross-process attacks (c.f., [3, 4, 15, 31, 32]) all of which target per-core microarchitectural state. When simultaneous multi-threading (SMT) is disabled (as is typical in cloud settings), such per-core attacks require that the attacker time-shares a CPU core with the victim. In order to obtain frequent observations of shared state, attacks abuse scheduler mechanisms that prioritize interactive workloads in order to preempt the victim. For example, ZJRR use inter-processor interrupts to preempt every 16 µs on average. In their cross-process attack, Bangerter et al. abuse the Linux process scheduler [15].

Fewer attacks thus far have abused (what we call) off-core state, such as last-level caches used by multiple cores. Some off-core attacks are coarse-grained, allowing attackers to learn only a few bits of information (e.g., whether the victim is using the cache or not [35]). An example of a fine-grained off-core attack is the recent Flush+Reload attack of Yarom and Falkner [45]. Their attack extends the Bangerter et al. attack to instead target last-level caches on some modern Intel processors and has been shown to enable very efficient theft of cryptographic keys in both cross-process and cross-VM settings. However, like the Bangerter et al. attack, it relies on the attacker and victim having shared memory pages. This is a common situation for cross-process settings, but also arises in cross-VM settings should the hypervisor perform memory page deduplication. While several hypervisors implement deduplication, thus far no IaaS clouds are known to use the feature and so are not vulnerable.

Threat model and goals. Our goal is to mitigate or completely prevent cross-VM attacks relevant to modern public IaaS cloud computing settings. We assume the attacker and victim are separate VMs co-resident on the same server running a Type I hypervisor. The attacker controls the entire VM, including the guest operating system and applications, but the hypervisor is run by the cloud provider and is trusted. SMT and memory deduplication are disabled. In this context, the best known attacks rely on:

(1) *Shared per-core state* that is accessible to the attacker and that has visibly different behavior based on its state, such as caches and branch predictors.

(2) *The ability to preempt the victim VM* at short intervals to allow only a few changes to that hardware state.

(3) *Access to a system clock* with enough resolution to distinguish micro-architectural events (e.g., cache hits and misses).

These conditions are all true in contemporary multi-tenant cloud settings, such as Amazon's EC2. Defenses can target any of these dependencies, and we discuss some existing approaches next.

Prior defenses. Past work on defenses against such side-channel attacks identified the above requirements for successful side-channels and tried to obviate one or more of the above necessary conditions for attacks. We classify and summarize these techniques below.

An obvious solution is to prevent an attacker and victim from sharing hardware, which we call *hard isolation*. Partitioning the cache in hardware or software prevents its contents from being shared [22, 34, 36, 42]. This requires special-purpose hardware or loss of various useful features (e.g., large pages) and thus limits the adoption in a public cloud environment. Assigning VMs to run on different cores avoids sharing of per-core hardware [21, 27, 38], and assigning them to different servers avoids sharing of any system hardware [35]. A key challenge here is identifying an attacker and victim in order to separate them; otherwise this approach reduces to using dedicated hardware for each customer, reducing utilization and thus raising the price of computing.

Another form of hard isolation is to reset hardware state when switching from one VM to another. For example, flushing the caches on every context switch prevents the cache state from being shared between VMs [47]. However, this can decrease performance of cache-sensitive workloads both because of the time taken to do the flush and the loss in cache efficiency.

Beyond hard isolation are approaches that modify hardware to add noise, either in the timing or by obfuscating the specific side-channel information. The former can be accomplished by removing or modifying timers [26, 28, 41] to prevent attackers from accurately distinguishing between microarchitectural events, such as a cache hit and a miss. For example, StopWatch [26] removes all timing side-channels and incurs a worse-case overhead of 2.8x for network intensive workloads. Specialized hardware-support could also be used to obfuscate and randomize processor cache usage [25, 44]. All

of these defenses either result in loss of high-precision timer or require hardware changes.

Similarly, one can allocate exclusive memory resources for a sensitive process [22] or add noise to obfuscate its hardware usage [47]. Similarly, programs can be changed to obfuscate access patterns [8, 9]. These approaches are not general-purpose, as they rely on identifying and fixing all security-relevant programs. Worst-case overheads for these mechanisms vary from 6–7%.

Several past efforts attempt to minimize performance interference between workloads (e.g., Q-clouds [30] and Bubble-Up [27], but do not consider adversarial workloads such as side-channel attacks.

3 MRT Guarantees and Soft Isolation

We investigate a different strategy for mitigating per-core side-channels: adjusting hypervisor core scheduling to limit the rate of preemptions. This targets the second requirement of attacks such as ZJRR. Such a scheduler would realize a security design principle that we call *soft isolation*[1]: limiting the frequency of potentially dangerous interactions between mutually untrustworthy programs. Unlike hard isolation mechanisms, we will allow shared state but attempt to use scheduling to limit the damage. Ideally, the flexibility of soft isolation will ease the road to deployment, while still significantly mitigating or even preventing side-channel attacks. We expect that soft isolation can be incorporated as a design goal in a variety of resource management contexts. That said, we focus in the rest of this work on CPU core scheduling.

Xen scheduling. Hypervisors schedule virtual machines much like an operating system schedules processes or threads. Just as a process may contain multiple threads that can be scheduled on different processors, a virtual machine may consist of multiple virtual CPUs (VCPUs) that can be scheduled on different physical CPUs (PCPUs). The primary difference between hypervisor and OS scheduling is that the set of VCPUs across all VMs is relatively static, as VM and VCPU creation/deletion is a rare event. In contrast, processes and threads are frequently created and deleted.

Hypervisor schedulers provide low-latency response times to interactive tasks by prioritizing VCPUs that need to respond to an outstanding event. The events are typically physical device or virtual interrupts from packet arrivals or completed storage requests. Xen's credit scheduler normally lets a VCPU run for 30ms before preempting it so another VCPU can run. However,

[1]The term "soft" is inherited from soft real-time systems, where one similarly relaxes requirements (in that case, time deadlines, in our case, isolation).

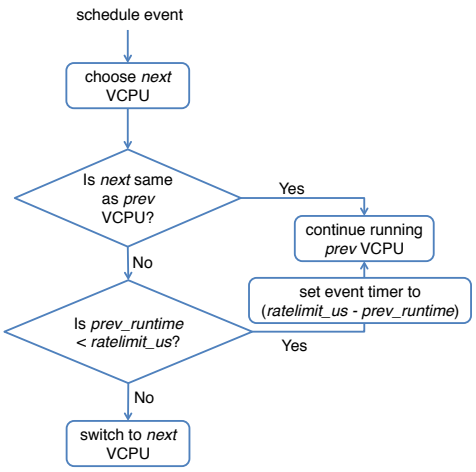


Figure 1: **Logic underlying the Xen MRT mechanism.**

when a VCPU receives an event, it may receive *boost* priority, which allows it to preempt non-boosted VCPUs and run immediately.

VCPUs are characterized by Xen as either *interactive* (or *latency-sensitive*) if they are mostly idle until an interrupt comes in, at which point they execute for a short period and return to idle. Typical interactive workloads are network servers that execute in response to an incoming packet. We refer to VCPUs that are running longer computations as *batch* or *CPU-hungry*, as they typically execute for longer than the scheduler's time slice (30ms for Xen) without idling.

Schedulers can be *work conserving*, meaning that they will never let a PCPU idle if a VCPU is ready to run, or *non-work conserving*, meaning that they enforce strict limits on how much time a VCPU can run. While work-conserving schedulers can provide higher utilization, they also provide worse performance isolation: if one VCPU goes from idle to CPU-hungry, another VCPU on the same PCPU can see its share of the PCPU drop in half. As a result, many cloud environments use non-work conserving schedulers. For example, Amazon EC2's *m1.small* instances are configured to be non-work conserving, allocating roughly 40% of a PCPU (called cap in Xen) to each VCPU of a VM.

Since version 4.2, Xen has included a mechanism for rate limiting preemptions of a VCPU; we call this mechanism a minimum run-time (MRT) guarantee. The logic underlying this mechanism is shown as a flowchart in Figure 1. Xen exposes a hypervisor parameter, `ratelimit_us` (the MRT value) that determines the minimum time any VCPU is guaranteed to run on a PCPU before being available to be context-switched out of the PCPU by another VCPU. One could also rate limit preemptions in other ways, but an MRT guarantee is sim-

ple to implement. Note that the MRT is not applicable to VMs that voluntarily give up the CPU, which happens when the VM goes idle or waits for an event to occur.

As noted previously, the original intent of Xen's MRT was to improve performance for CPU-hungry workloads run in the presence of latency-sensitive workloads: each preemption pollutes the cache and other microarchitectural state, slowing the CPU-intensive workload

Case study. We experimentally evaluate the Xen MRT mechanism as a defense against side-channel leakage by way of soft isolation. Intuitively, the MRT guarantee rate-limits preemptions and provides an attacker less granularity in his observations of the victim's use of per-CPU-core resources. Thus one expects that increased rate-limits decreases vulnerability. To be deployable, however, we must also evaluate the impact of MRT guarantees on benign workloads. In the next two sections we investigate the following questions:

(1) How do per-core side-channel attacks perform under various MRT values? (Section 4)

(2) How does performance vary with different MRT values? (Section 5)

4 Side-channels under MRT Guarantees

We experimentally evaluate the MRT mechanism as a defense against side-channel leakage for per-core state. We focus on cache-based leakage.

Experimental setup. Running on the hardware setup shown in Figure 2, we configure Xen to use two VMs, a victim and attacker. Each has two VCPUs, and we pin one attacker VCPU and one victim VCPU to each of two PCPUs (or cores). We use a non-work-conserving scheduler whose configuration is shown in Figure 9. This is a conservative version of the ZJRR attack setting, where instead the VCPUs were allowed to float — pinning the victims to the same core only makes it easier for the attacker. The hardware and Xen configurations are similar to the configuration used in EC2 m1.small instances [12]. (Although Amazon does not make their precise hardware configurations public, we can still gain some insight into the hardware on which an instance is running by looking at `sysfs` and the `CPUID` instruction.)

Cache-set timing profile. We start by fixing a simple victim to measure the effects of increasing MRT guarantees. We have two functions that each access a (distinct) quarter of the instruction cache (I-cache)[2]. The victim

[2]Our test machine has a 32 KB, 4-way set associative cache with 64-byte lines. There are 128 sets.

Machine Configuration	Intel Xeon E5645, 2.40GHz clock, 6 cores in one package
Memory Hierarchy	Private 32 KB L1 (I- and D-cache), 256 KB unified L2, 12 MB shared L3 and 16 GB main memory.
Xen Version	4.2.1
Xen Scheduler	Credit Scheduler 1
Dom0 OS	Fedora 18, 3.8.8-202.fc18.x86_64
Guest OS	Ubuntu 12.04.3, Linux 3.7.5

Figure 2: **Hardware configuration in local test bed.**

alternates between these two functions, accessing each quarter 500 times. This experiment models a simple I-cache side-channel where switching from one quarter to another leaks some secret information (we call any such leaky function a *sensitive* operation). Executing the 500 access to a quarter of the I-cache requires approximately 100 µs when run in isolation.

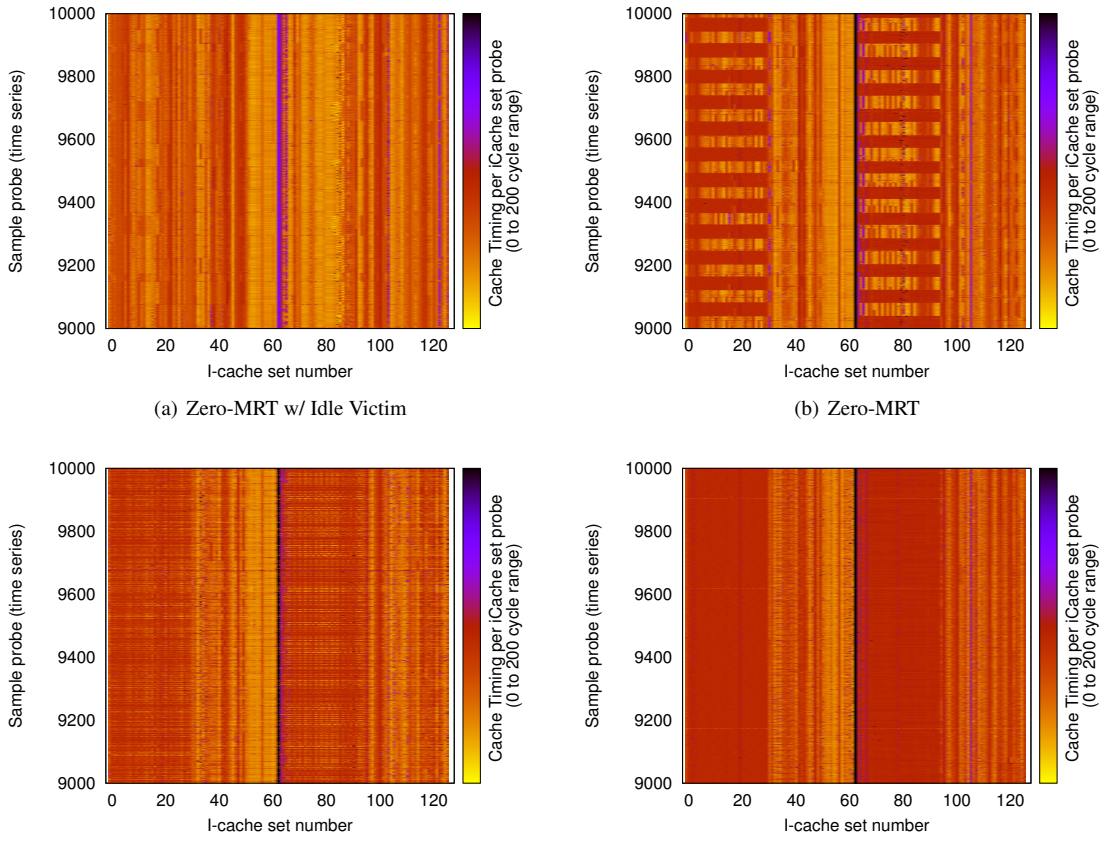
We run this victim workload pinned to a victim VCPU that is pinned to the same PCPU as the attacker VCPU. The attacker uses the IPI-based Prime+Probe technique[3] and measures the time taken to access each I-cache set, similar to ZJRR [46].

Figure 3 shows heat maps of the timings of the various I-cache sets as taken by the Prime+Probe attacker, for various MRT values between 0 (no MRT) and 5 ms. Darker colors are longer access times, indicating conflicting access to the cache set by the victim. One can easily see the simple alternating pattern of the victim as we move up the y-axis of time in Figure 3(b). Also note that this is different from an idle victim under zero-MRT shown in Figure 3(a). With no MRT, the attacker makes approximately 40 observations of each cache set, allowing a relatively detailed view of victim behavior.

As the MRT value increases we see the loss of resolution by the attacker as its observations become less frequent than the alternations of the victim. At an MRT of 100 µs the pattern is still visible, but noisier. Although the victim functions run for 100 µs, the prime+probe attacker slows downs the victim by approximately a factor of two, allowing the pattern to be visible with a 100 µs MRT. When the MRT value is set to 1 ms the attacker obtains no discernible information on when the switching between each I-cache set happens.

In general, an attacker can observe victim behavior that occurs at a lower frequency than the attacker's preemptions. We modify the victim program to be 10x

[3]Note that the attacker requires two VCPUs, one measuring the I-cache set timing whenever interrupted and the other issuing the IPIs to wake up the other VCPU. The VCPU issuing IPIs is pinned to a different PCPU.



(a) Zero-MRT w/ Idle Victim

(b) Zero-MRT

(c) 100 µs-MRT

(d) 1 ms-MRT

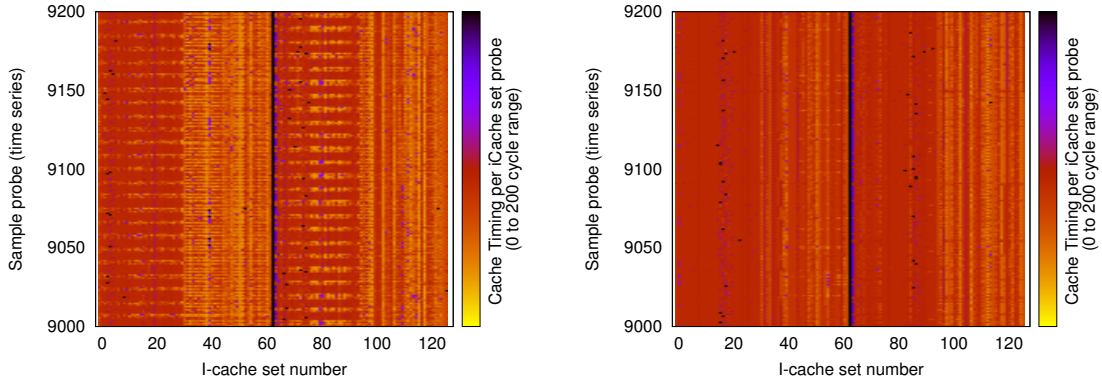
Figure 3: **Heatmaps of I-cache set timing as observed by a prime-probe attacker.** Displayed values are from a larger trace of 10,000 timings. (a) Timings for idle victim and no MRT. (b)–(d) Timings for varying MRT values with the victim running.

slower (where each function takes approximately 1 ms standalone). Figure 4 shows the result for this experiment. With a 1 ms MRT, we observe the alternating pattern. When the MRT is raised to 5 ms, which is longer than the victim's computation ($\approx$ 2 ms), no pattern is apparent. Thus, when the MRT is longer than the execution time of a security-critical function this side-channel fails.

While none of this proves lack of side-channels, it serves to illustrate the dynamics between side-channels, duration of sensitive victim operations, and the MRT: as the MRT increases, the frequency with which an attacker can observe the victim's behavior decreases, and the signal and hence leaked information decreases. All this exposes the relationship between the speed of a sensitive operation, the MRT, and side-channel availability for an attacker. In particular, very long operations (e.g., longer than the MRT) may still be spied upon by side-channel attackers. Also, infrequently accessed but sensitive memory accesses may leak to the attacker. We hypothesis that at least for cryptographic victims, even moderate MRT values on the order of a handful of milliseconds are sufficient to prevent per-core side-channel attacks. We next look, therefore, at how this relationship plays out for cryptographic victims.

ElGamal victim. We fix a victim similar to that targeted by ZJRR. The victim executes the modular exponentiation implementation from libgcrypt 1.5.0 using a 2048-bit exponent, base and modulus, in a loop. Pseudocode of the exponentiation algorithm appears in Figure 5. One can see that learning the sequence of operations leaks the secret key values: if the code in lines 7 and 8 is executed, the bit is a 1; otherwise it is a zero. We instrument libgcrypt to write the current bit being operated upon to a memory page shared with the attacker, allowing us to determine when preemptions occur relative to operations within the modular exponentiation.

For no MRT guarantee, we observe that the attacker can preempt the victim many times per individual square, multiply, or reduce operation (as was also reported by ZJRR). With MRT guarantees, the rate of preemptions drops so much that the attacker only can interrupt once



(a) 1 ms-MRT (b) 5 ms-MRT

Figure 4: **Heatmaps of I-cache set timings as observed by a prime-probe attacker for 10x slower victim computations.** Displayed values are from a larger trace of 9,200 timings.

```
SQUAREMULT(x, e, N):
 1: Let e_n, ..., e_1 be the bits of e
 2: y ← 1
 3: for i = n down to 1 do
 4:     y ← SQUARE(y)
 5:     y ← MODREDUCE(y, N)
 6:     if e_i = 1 then
 7:         y ← MULT(y, x)
 8:         y ← MODREDUCE(y, N)
 9:     end if
10: end for
11: return y
```

Figure 5: **Modular exponentiation algorithm used in libgcrypt version 1.5.0.** Note that the control flow followed when $e_i = 1$ is lines $4 \to 5 \to 6 \to 7 \to 8$ and when $e_i = 0$ is lines $4 \to 5$; denoted by the symbols `1` and `0`, respectively.

Xen MRT (ms)	Avg. ops/run	Min. ops/run
0	0.096	0
0.1	14.1	4
0.5	49.0	32
1.0	92.6	68
2.0	180.7	155
5.0	441.2	386
10.0	873.1	728

Figure 6: **The average and minimal number of ElGamal secret key bits operated upon between two attacker preemptions for a range of MRT values.** Over runs with 40K preemptions.

Xen MRT (ms)	Preemptions per function call		
	Min	Median	Max
0	3247	19940	20606
0.1	74	155	166
0.5	22	42	47
1.0	16	22	25
2.0	10	11	13
5.0	0	4	6
10.0	1	2	3

Figure 7: **Rate of preemption with various MRT.** Here the function called is the Modular-Exponentiation implementation in libgcrypt with a 2048 bit exponent. Note that for zero MRT the rate of preemption is very high that victim computation involving a single bit was preempted multiple times.

every several iterations of the inner loop. Figure 6 gives the number of bits operated on between attacker preemptions for various MRT values. Figure 7 gives the number of preemptions per entire modular exponentiation computation. We see that for higher MRT values, the rate of preemption per call to the full modular exponentiation reduces to just a handful. The ZJRR attack depends on multiple observations *per operation* to filter out noise, so even at the lowest MRT value of 100 µs, with 4–14 operations per observation, the ZJRR attack fails. In the full version [40], we discuss how one might model this leakage scenario formally and evidence a lack of any of a large class of side-channel attacks.

AES victim. We evaluate another commonly exploited access-driven side-channel victim, AES, which leaks secret information via key-dependent indexing into tables stored in the L1 data cache [15, 31]. The previous attacks, all in the cross-process setting, depend on observing a very small number of cache accesses to obtain a clear signal of what portion of the table was accessed by the victim. Although there has been no known AES attack in the cross-VM setting (at least when deduplication is turned off, otherwise see [19]), we evaluate effectiveness of MRT against the best known IPI Prime+Probe

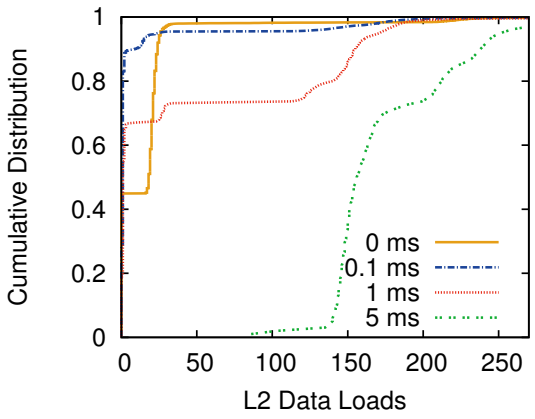


Figure 8: **CDF of L2 data-loads per time slice experienced by OpenSSL-AES victim.** L2 data loads are performance counter events that happen when a program requests for a memory word that is not in both L1 and L2 private caches (effectively a miss). When running along-side a Prime+Probe attacker, these data-cache accesses can be observed by the attacker.

spy process due to ZJRR. In particular, we measured the number of private data-cache misses possibly observable by this Prime+Probe attacker when the victim is running AES encryption in a loop.

To do so, we modified the Xen scheduler to log the count of private-cache misses (in our local testbed both L1 and L2 caches are private) experienced by any VCPU during a scheduled time slice. This corresponds to the number of data-cache misses an attacker could ideally observe. Figure 8 shows the cumulative distribution of the number of L2-data cache misses (equivalently, private data-cache loads) during a time slice of the victim running OpenSSL-AES. We can see that under no or lower MRTs the bulk of time slices suffer only a few tens of D-cache misses that happen between two back-to-back preemptions of the attacker. (We note that this is already insufficient to perform prior attacks.) The number of misses increases to close to 200 for an MRT value of 5 ms. This means that the AES process is evicting its own data, further obscuring information from a would-be attacker. Underlying this is the fact that the number of AES encryptions completed between two back-to-back preemptions increases drastically with the MRT: found that thousands to ten thousands AES block-encryptions were completed between two preemptions when MRT was varied from 100 µs to 5 ms, respectively.

Summary. While side channels pose a significant threat to the security of cloud computing, our measurements in this section show that, fortunately, the hypervisor scheduler can help. Current attacks depend on frequent preemptions to make detailed measurements of cache contents. Our measurements show that even delaying preemption for a fraction of millisecond prevents known attacks. While this is not proof that future attacks won't be found that circumvent the MRT guarantee, it does strongly suggest that deploying such a soft-isolation mechanism will raise the bar for attackers. This leaves the question of whether this mechanism is cheap to deploy, which we answer in the next section.

Note that we have focused on using the MRT mechanism for CPU and, indirectly, per-core hardware resources that are shared between multiple VMs. But rate-limiting-type mechanisms may be useful for other shared devices like memory, disk/SSD, network, and any system-level shared devices which suffer from a similar access-driven side-channels. For instance, a timed disk read could reveal user's disk usage statistics like relative disk head positions [20]. Fine-grained sharing of the disk across multiple users could leak sensitive information via such a timing side-channel. Reducing the granularity of sharing by using MRT-like guarantees in the disk scheduler (e.g., servicing requests from user for at least T_{min}, minimum service time, before serving requests from another user) would result in a system with similar security guarantees as above, eventually making such side-channels harder to exploit. Further research is required to analyze the end-to-end performance impact of such a mechanism for various shared devices and schedulers that manage them.

5 Performance of MRT Mechanism

The analysis in the preceding section demonstrates that MRT guarantees can meaningfully mitigate a large class of cache-based side-channel attacks. The mitigation becomes better as MRT increases. We therefore turn to determining the maximal MRT guarantee one can fix while not hindering performance.

5.1 Methodology

We designed experiments to quantify the negative and positive effects of MRT guarantees as compared to a baseline configuration with no MRT (or zero MRT). Our testbed configuration uses the same hardware as in the last section and the Xen configurations are summarized in Figure 9. We run two DomU VMs each with a single VCPU. The two VCPUs are pinned to the same PCPU. Pinning to the same PCPU serves to isolate the effect of the MRT mechanism. The management VM, Dom0, has 6 VCPUs, one for each PCPU (a standard configuration option). The remaining PCPUs in the system are otherwise left idle.

Work-conserving configuration

Dom0	6 VCPU / no cap / weight 256
DomU	1 VCPU / 2 GB memory / no cap / weight 256

Non-work-conserving configuration

Dom0	6 VCPU / no cap / weight 512
DomU	1 VCPU / 2 GB memory / 40% cap / weight 256

Figure 9: **Xen configurations used for performance experiments.**

CPU-hungry Workloads

Workload	Description
SPECjbb	Java-based application server [37]
graph500	Graph analytics workload [1] with scale of 18 and edge factor of 20.
mcf, *sphinx*, *bzip2*	SpecCPU2006 cache sensitive benchmarks [17]
Nqueens	Microbenchmark solving n-queens problem
CProbe	Microbenchmark that continuously trashes L2 private cache.

Latency-sensitive Workloads

Workload	Description
Data-Caching	Memcached from Cloud Suite-2 with twitter data set scaled by factor of 5 run for 3 minutes with rate of 500 requests per second [13].
Data-Serving	Cassandra KV-store from Cloud Suite-2 with total of 100K records[4] [13]
Apache	Apache webserver, HTTPing client [18], single 4 KB file at 1 ms interval.
Ping	Ping command at 1 ms interval.
Chatty-CProbe	One iteration of *CProbe* every 10 µs.

Figure 10: **Workloads used in performance experiments.**

We use a mix of real-world applications and microbenchmarks in our experiments (shown in Figure 10). The microbenchmark *CProbe* simulates a perfectly cache-sensitive workload that continuously overwrites data to the (unified) L2 private cache, and *Chatty-CProbe* is its interactive counterpart that overwrites the cache every 10 µs and then sleeps. We also run the benchmarks with an idle VCPU (labeled *Idle* below).

5.2 Latency Sensitivity

The most obvious potential performance downside of a MRT guarantee is increased latency: interactive workloads may have to wait before gaining access to a PCPU. We measure the negative effects of MRT guarantees by running latency-sensitive workloads against *Nqueens* (a CPU-bound program with little memory access). Figure 11 shows the 95^{th} percentile latency for the interactive workloads. The baseline results are shown as a MRT of 0 on the X-axis. As expected, the latency is approximately equal to the MRT for almost all workloads (*Apache* has higher latency because it requires multiple packets to respond, so it must run multiple times to complete a request). Thus, in the presence of a CPU-intensive workload and when pinned to the same PCPU, the MRT can have a large negative impact on interactive latency.

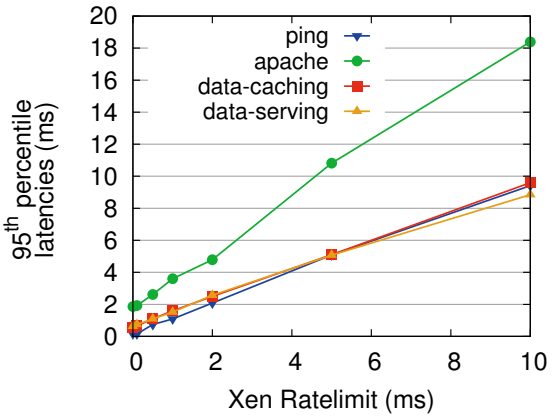


Figure 11: **95^{th} Percentile Latency of Various Latency Sensitive Workloads.** Under non-work-conserving scheduling.

As the workloads behave essentially similarly, we now focus on just the *Data-Caching* workload. Figure 12 shows the response latency when run against other workloads. For the two CPU-intensive workloads, *CProbe* and *Nqueens*, latency increases linearly with the MRT. However, when run against either an idle VCPU or *Chatty-CProbe*, which runs for only a short period, latency is identical across all MRT values. Thus, the MRT has little impact when an interactive workload runs alone or it shares the PCPU with another interactive workload.

We next evaluate the extent of latency increase. Figure 13 shows the 25^{th}, 50^{th}, 75^{th}, 90^{th}, 95^{th} and 99^{th} percentile latency for *Data-Caching*. At the 50^{th} percentile and below, latency is the same as with an idle VCPU. However, at the 75^{th} latency rises to half the MRT, indicating that a substantial fraction of requests are delayed.

We repeated the above experiments for the work-conserving setting, and the results were essentially the same. We omit them for brevity. Overall, we find that enforcing an MRT guarantee can severely increase latency when interactive VCPUs share a PCPU with CPU-intensive workloads. However, they have limited impact when multiple interactive VCPUs share a PCPU.

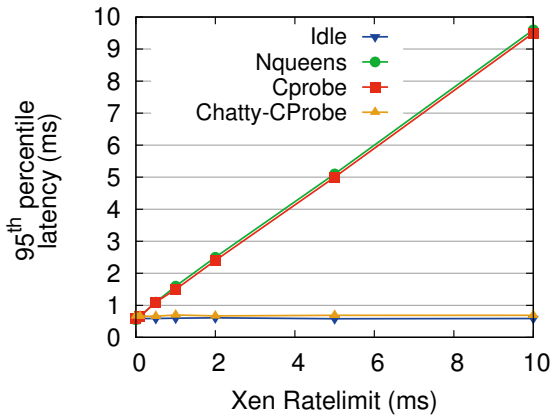


Figure 12: **95th Percentile Request Latency of *Data-Caching* Workload with Various Competing Micro-benchmarks.** Under non-work-conserving scheduling.

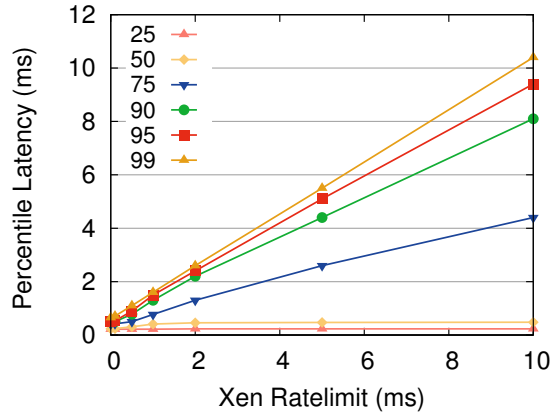


Figure 13: **25th, 50th, 75th, 90th, 95th and 99th Percentile Latency of *Data-Caching* Workload when run alongside *Nqueens*.** Under non-work-conserving scheduling.

5.3 Batch Efficiency

In addition to measuring the impact on latency-sensitive workloads, we also measure the impact of MRT guarantees on CPU-hungry workloads. The original goal of the MRT mechanism was to reduce frequent VCPU context-switches and improve performance of batch workloads. We pin a CPU-hungry workload to a PCPU against competing microbenchmarks.

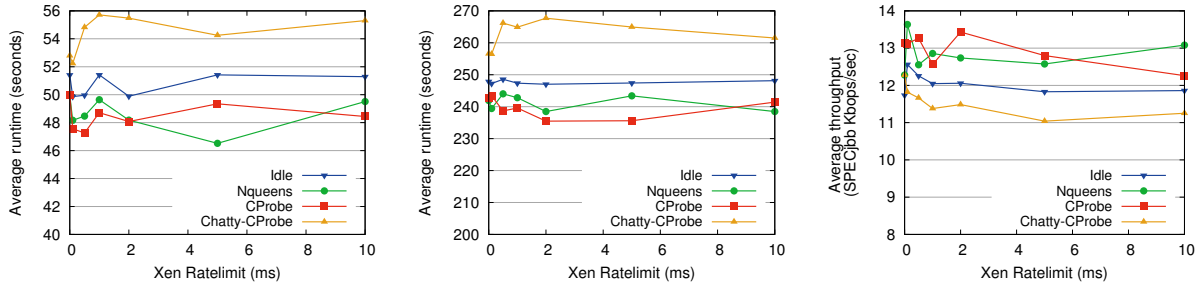
Figure 14 shows the effect of MRT values on the *graph500* workload when run alongside various competing workloads. Because this is work-conserving scheduling, the runtime of *graph500* workload increases by roughly a factor of two when run alongside *Nqueens* and *CProbe* as compared to *Idle*, because the share of the PCPU given to the VCPU running *graph500* drops by one half. The affect of MRT is more pronounced when looking running alongside *Chatty-CProbe*, the workload which tries to frequently interrupt *graph500* and trash its cache. With no MRT guarantee, this can double the runtime of a program. But with a limit of only 0.5 ms, performance is virtually the same as with an idle VCPU, both because *Chatty-CProbe* uses much less CPU and because it trashes the cache less often.

With a non-work-conserving scheduler, the picture is significantly different. Figure 15 shows the performance of three batch workloads when run alongside a variety of other workloads, for various MRT values. First, we observe that competing CPU-bound workloads such as *Nqueens* and *CProbe* do not significantly affect the performance of CPU-bound applications, even in the case of *CProbe* that trashes the cache. This occurs because the workloads share the PCPU at coarse intervals (30 ms), so the cache is only trashed once per 30 ms period. In contrast, when run with the interactive workload *Chatty-CProbe*, applications suffer up to 4% performance loss, which *increases* with longer MRT guarantees. Investigating the scheduler traces showed that under zero MRT the batch workload enjoyed longer scheduler time slices of 30 ms compared to the non-zero MRT cases. This was because under zero MRT highly interactive *Chatty-CProbe* quickly exhausted Xen's boost priority. After this, *Chatty-CProbe* could not preempt and waited until the running VCPU's 30 ms time slice expires. With longer MRT values, though, *Chatty-CProbe* continues to preempt and degrade performance more consistently.

Another interesting observation in Figure 15 is that when the batch workloads share a PCPU with an idle VCPU, they perform worse than when paired with *Nqueens* or *CProbe*. Further investigation revealed that an idle VCPU is not completely idle but wakes up at regular intervals for guest timekeeping reasons. Overall, under non-work-conserving settings, running a batch VCPU with any interactive VCPU (even an idle one) is worse than running with another batch VCPU (even one like *CProbe* that trashes the cache).

5.4 System Performance

The preceding sections showed the impact of MRT guarantees when both applications are pinned to a single core. We next analyze the impact of the Xen scheduler's VCPU placement policies, which choose the PCPU on which to schedule a runnable VCPU. We configure the system with 4 VMs each with 2 VCPUs to run on 4 PCPUs under a non-work conserving scheduler. We run three different sets of workload mixes, which together capture a broad spectrum of competing workload combinations. Together with a target workload running on both VCPUs of a single VM, we run: (1) *All-Batch* — consisting of worst-case competing CPU-hungry work-



(a) *mcf* (b) *graph500* (c) *SPECjbb*

Figure 15: **Average runtime of various batch workloads under non-work conserving setting.** Note that for *SPECjbb* higher is better (since the graph plots the throughput instead of runtime). All data points are averaged across 5 runs.

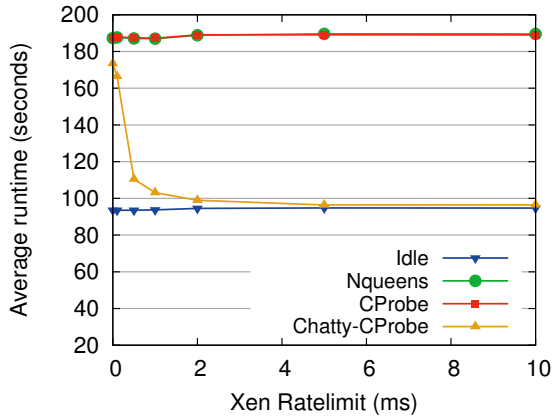


Figure 14: **Average runtime of *graph500* workload when run alongside various competing workloads and under work-conserving scheduling.** Averaged over 5 runs.

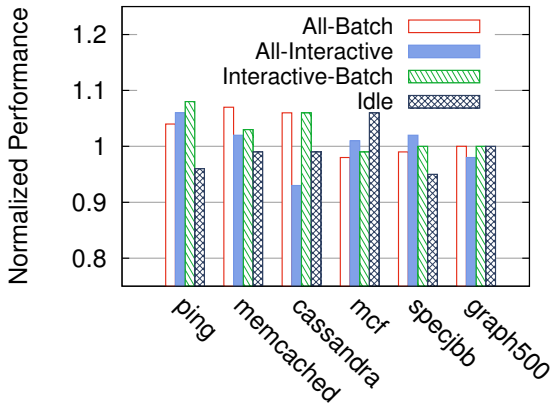


Figure 16: **Normalized Performance in a non-work-conserving configuration with 5 ms MRT.** Normalized to performance under zero-MRT case. The left three workloads report 95^{th} percentile latency and the right three report runtime, averaged across 5 runs. In both cases lower is better.

load (*CProbe*); (2) *All-Interactive* — consisting of worst-case competing interactive workload (*Chatty-CProbe*); and (3) *Batch & Interactive* — where half of other VCPUs run *Chatty-CProbe* and half *CProbe*. We compare the performance of Xen without MRT to running with the default 5ms limit. The result of the experiment is shown in Figure 16. For interactive workloads, the figure shows the relative 95^{th} percentile latency, while for CPU-hungry workloads it shows relative execution time.

On average across the three programs and three competing workloads, latency-sensitive workloads suffered on average of only 4% increase in latency with the MRT guarantee enabled. This contrasts sharply with the 5-fold latency increase in the pinned experiment discussed earlier. CPU-hungry workloads saw their performance improve by 0.3%. This makes sense given the results in the preceding section, which showed that an MRT guarantee offers little value to batch jobs in a non-work-conserving setting.

To understand why the latency performance is so much better than our earlier results would suggest, we analyzed a trace of the scheduler's decisions. With the non-work-conserving setting, Xen naturally segregates batch and interactive workloads. When an interactive VCPU receives a request, it will migrate to an idle PCPU rather than preempt a PCPU running a batch VCPU. As the PCPU running interactive VCPUs is often idle, this leads to coalescing the interactive VCPUs on one or more PCPUs while the batch VCPUs share the remaining PCPUs.

5.5 Summary

Overall, our performance evaluation shows that the strong security benefits described the in Section 4 can be achieved at low cost in virtualized settings. Prior research suggests more complex defense mechanisms [22, 26, 28, 43, 47] that achieve similar low performance overheads but at a higher cost of adoption, such as sub-

stantial hardware changes or modifications to security-critical programs. In comparison, the MRT guarantee mechanism is simple and monotonically improves the security against many existing side-channel attacks with zero cost of adoption and low overhead.

We note that differences between hypervisor scheduling and OS scheduling mean that the MRT mechanism cannot be as easily applied by an operating system to defend against malicious processes. As mentioned above, a hypervisor schedules a small and relatively static number of VCPUS onto PCPUs. Thus, it is feasible to coalesce VCPUs with interactive behavior onto PCPUs separate from those running batch VCPUs. Furthermore, virtualized settings generally run with *share-based* scheduling, where each VM or VCPU is assigned a fixed share of CPU resources. In contrast, the OS scheduler must schedule an unbounded number of threads, often without assigned shares. Thus, there may be more oversubscription of PCPUs, which removes the idle time that allows interactive VCPUs to coalesce separately from batch VCPUs. As a result, other proposed defenses may still be applicable for non-virtualized systems, such as PaaS platforms that multiplex code from several customers within a single VM [2].

6 Integrating Core-State Cleansing

While the MRT mechanism was shown to be a cheap mitigation for protecting CPU-hungry workloads, it may not be effective at protecting interactive ones. If a (victim) VCPU yields the PCPU quickly, the MRT guarantee does not apply and an attacker may observe its residual state in the cache, branch predictor, or other hardware structures. We are unaware of any attacks targeting such interactive workloads, but that is no guarantee future attacks won't.

We investigate incorporating per-core state-cleansing into hypervisor scheduling. Here we are inspired in large part by the Düppel system [47], which was proposed as a method for guest operating systems to protect themselves by periodically cleansing a fraction of the L1 caches. We will see that by integrating a selective state-cleansing (SC) mechanism for I-cache, D-cache and branch predictor states into a scheduler that already enforces an MRT guarantee incurs much less overhead than one might expect. When used, our cleansing approach provides protection for all processes within a guest VM (unlike Düppel, which targeted particular processes).

6.1 Design and Implementation

We first discuss the cleansing process, and below discuss when to apply it. The cleanser works by executing a specially crafted sequence of instructions that together overwrite the I-cache, D-cache, and branch predictor states of a CPU core. A sample of these instructions is shown in Figure 17; these instructions are 27 bytes long and fit in a single I-cache line.

In order to overwrite the branch predictor or the Branch Target Buffer (BTB) state, a branch instruction conditioned over a random predicate in memory is used. There are memory move instructions that add noise to the D-cache state as well. The last instruction in the set jumps to an address that corresponds to the next way in the same I-cache set. This jump sequence is repeated until the last way in the I-cache set is accessed, at which point it is terminated with a `ret` instruction. These instructions and the random predicates are laid out in memory buffers that are equal to the size of the I-cache and D-cache, respectively. Each invocation of the cleansing mechanism randomly walks through these instructions to touch all I-cache sets, D-cache sets, and flush the BTB.

We now turn to how we have the scheduler decide when to schedule cleansing. There are several possibilities. The simplest strategy would be to check, when a VCPU wakes up, if the prior running VCPU was from another VM and did not use up its MRT. If so, then run the cleansing procedure before the incoming VCPU. We refer to this strategy as *Delayed-SC* because we defer cleansing until a VCPU wants to execute. This strategy guarantees to cleanse only when needed, but has the downside of potentially hurting latency-sensitive applications (since the cleanse has to run between receiving an interrupt and executing the VCPU). Another strategy is to check, when a VCPU relinquishes the PCPU before its MRT guarantee expires, whether the next VCPU to run is from another domain or if the PCPU will go idle. In either case, a cleansing occurs before the next VCPU or idle task runs. Note that we may do unnecessary cleansing here, because the VCPU that runs after idle may be from the same domain. We therefore refer to this strategy as *Optimistic-SC*, given its optimism that a cross-VM switch will occur after idle. This optimism may pay off because idle time can be used for cleansing.

Note that the CPU time spent in cleansing in Delayed-SC is accounted to the incoming VCPU but it is often free with Optimistic-SC as it uses idle time for cleansing when possible.

6.2 Evaluation

We focus our evaluation on latency-sensitive tasks: because we only cleanse when an MRT guarantee is not hit, CPU-hungry workloads will only be affected minimally by cleansing. Quantitatively the impact is similar to the results of Section 5 that show only slight degradation due to *Chatty-CProbe* on CPU-hungry workloads.

We use the hardware configuration shown in Figure 2.

```
000 <L13-0xd>:
  0: 8b 08           mov    (%rax),%ecx
  2: 85 c9           test   %ecx,%ecx
  4: 74 07           je     d <L13>
  6: 8b 08           mov    (%rax),%ecx
  8: 88 4d ff        mov    %cl,-0x1(%rbp)
  b: eb 05           jmp    12 <L14>

00d <L13>:
  d: 8b 08           mov    (%rax),%ecx
  f: 88 4d ff        mov    %cl,-0x1(%rbp)

012 <L14>:
 12: 48 8b 40 08     mov    0x8(%rax),%rax
 17: e9 e5 1f 00 00  jmpq   <next way in set>
```

Figure 17: **Instructions used to add noise.** The assembly code is shown using X86 GAS Syntax. `%rax` holds the address of the random predicate used in the `test` instruction at the relative address `0x2`. The moves in the basic blocks `<L13>` and `<L14>` reads the data in the buffer, which uses up the corresponding D-cache set.

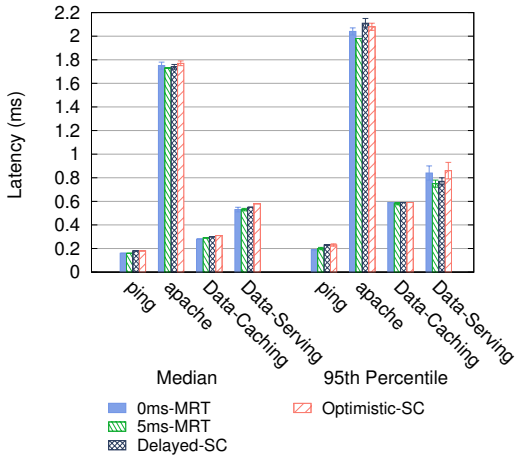


Figure 18: **Median and 95^{th} percentile latency impact of the cleansing scheduler under *worst-case* scenario.** Here all the measured workloads are feed by a client at 500 requests per second. The error bars show the standard deviation across 3 runs.

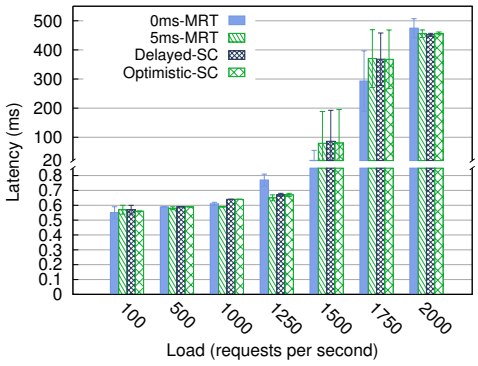
We measured the standalone, steady state execution time of the cleansing routine as 8.4 µs; all overhead beyond that is either due to additional cache misses that the workload experiences or slow down of the execution of the cleansing routine which might itself experience additional cache misses. To measure the overhead of the cleansing scheduler, we pinned two VCPUs of two different VMs to a single PCPU. We measured the performance of one of several latency-sensitive workloads running within one of these VMs, while the other VM ran a competing workload similar to *Chatty-CProbe* (but it did not access memory buffers when awoken). This ensured frequent cross-VM VCPU-switches simulating a worst case scenario for the cleansing scheduler.

We ran this experiment in four settings: no MRT guarantee (0ms-MRT), a 5 ms MRT guarantee (5ms-MRT), a 5 ms MRT with Delayed-SC, and finally a 5 ms MRT with Optimistic-SC. Figure 18 shows the median and 95^{th} percentile latencies under this experiment. The median latency increases between 10–50 µs compared to the 5ms-MRT baseline, while the 95^{th} percentile results are more variable, and show at worst a 100 µs increase in tail latency. For very fast workloads, like *Ping*, this results in a 17% latency increase despite the absolute overhead being small. Most of the overhead comes from reloading data into the cache, as only 1/3rd of the overhead is from executing the cleansing code.

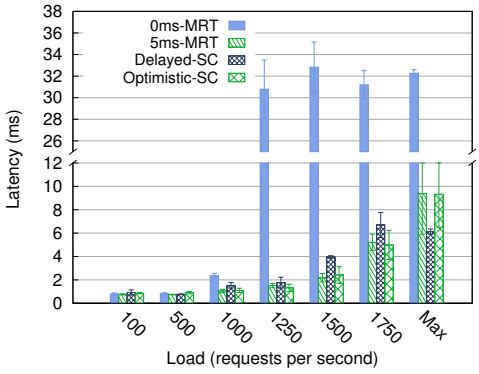
To measure overhead for non-adversarial workloads, we replaced the synthetic worst-case interactive workload with a moderately loaded Apache webserver (at 500 requests per second). The result of this experiment is not shown here as it looks almost identical to Figure 18, suggesting the choice of competing workload has relatively little impact on overheads. In this average-case scenario, we observed an overhead of 20–30 µs across all workloads for the *Delayed-SC* and 10–20 µs for *Optimistic-SC*, which is 10 µs faster. Note that in all the above cases, the cleansing mechanism perform better than the baseline of no MRT guarantee with no cleansing.

To further understand the trade-off between the two variations of state-cleansing, we repeated the first (worst-case) experiment above with varying load on the two latency-sensitive workloads, *Data-Caching* and *Data-Serving*. The 95^{th} percentile and median latencies of these workloads under varying loads are shown in Figure 19 and Figure 20, respectively. The offered load shown on the x-axis is equivalent to the load perceived at the server in all cases except for *Data-Serving* workload whose server throughput saturates at 1870rps (this is denoted as *Max* in the graph).

The results show that the two strategies perform similarly in most situations, with optimization benefiting in a few cases. In particular, we see that the 95% latency for heavier loads on *Data-Serving* (1250, 1500, and 1750) is significantly reduced for Optimistic-SC over Delayed-SC. It turned out that the use of idle-time for cleansing in Optimistic-SC was crucial for *Data-Serving* workload as the tens to hundreds of microsecond overhead of cleansing mechanism under the Delayed-SC scheme was enough to exhaust boost priority at higher loads. From scheduler traces of the runs with *Data-Serving* at 1500rps, we found that the VM running the *Data-Serving* workload spent 1.9s without boost priority under Delayed-SC compared to 0.8s and 1.1s spent under

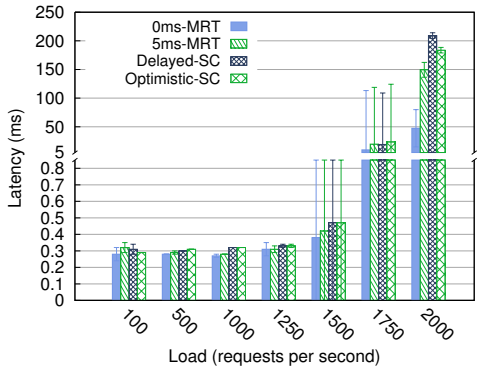


(a) *Data-Caching*

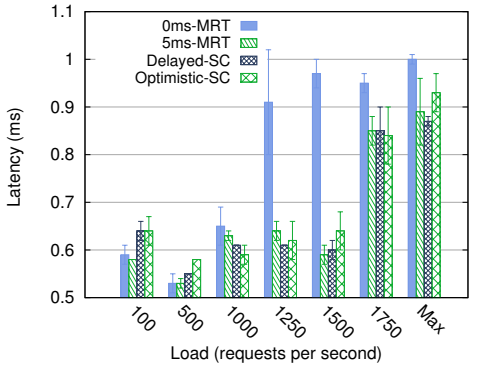


(b) *Data-Serving*

Figure 19: 95^{th} **percentile latency impact of the cleansing scheduler with varying load on the server.** (a) *Data-Caching* and (b) *Data-Serving*. The error bars show the standard deviation across 3 runs.



(a) *Data-Caching*



(b) *Data-Serving*

Figure 20: **Median latency impact of the cleansing scheduler with varying load on the servers.** (a) *Data-Caching* and (b) *Data-Serving*. The error bars show the standard deviation across 3 runs.

5ms-MRT and Optimistic-SC, respectively (over a 120 long second run). The *Data-Serving* VM also experienced 37% fewer wakeups under Delayed-SC relative to 5ms-MRT baseline, implying less interactivity.

We conclude that both strategies provide a high-performance mechanism for selectively cleansing, but that Optimistic-SC handles certain cases slightly better due to taking advantage of idle time.

7 Conclusions

Cloud computing promises improved efficiency, but opens up new threats due to the sharing of hardware across mutually distrustful customers. While virtual machine managers effectively prevent *direct access* to the data of other customers, current hardware platforms inherently leak information when that data is accessed through predictive structures such as caches and branch predictors.

We propose that the first line of defense against these attacks should be the software responsible for determining access: the hypervisor scheduler. For cache-based side channels, we showed that the simple mechanism of MRT guarantees can prevent useful information from being obtained via side-channel attacks that abuse per-core state. This suggests a high performance way of achieving soft isolation, which limits the frequency of potentially dangerous cross-VM interactions.

We also investigate how the classic defense technique of CPU state cleansing can interoperate productively with MRT guarantees. This provides added protection for interactive workloads at low cost, and takes advantage of the fact that the use of MRT makes rescheduling (each of which may require cleansing) rarer.

Finally we note that while the focus of our work was on side-channel attacks, the soft-isolation approach, and the mechanisms we consider in this paper in particular, should also be effective at mitigating other classes of shared-resource attacks. For example, resource-freeing [39] and on-system degradation-of-service attacks.

Acknowledgments

We thank Yinqian Zhang for sharing his thoughts on defenses against side-channel attacks and the code for the IPI-attacker that was used in the ZJRR attack. This work was supported in part by NSF grants 1253870, 1065134, and 1330308 and a generous gift from Microsoft. Swift has a significant financial interest in Microsoft.

References

[1] Graph 500 benchmark 1. `http://www.graph500.org/`.

[2] Heroku PaaS system. `https://www.heroku.com/`.

[3] O. Aciiçmez. Yet another microarchitectural attack:: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. ACM.

[4] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[5] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286, Feb. 2007.

[6] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 103–108. ACM, 2010.

[7] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[8] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of aes. In *Selected Areas in Cryptography*, pages 69–83. Springer, 2005.

[9] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[11] G. Dunlap. Xen 4.2: New scheduler parameters. http://blog.xen.org/index.php/2012/04/10/xen-4-2-new-scheduler-parameters-2/.

[12] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. ACM, 2012.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12. ACM, 2012.

[14] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 251–261, May 2001.

[15] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, 2011.

[16] M. W. B. Heinz and F. Stumpf. A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*, Feb. 2012.

[17] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[18] HTTPing. Httping client. `http://www.vanheusden.com/httping/`.

[19] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. Cryptology ePrint Archive, Report 2014/435, 2014. `http://eprint.iacr.org/`.

[20] P. A. Karger and J. C. Wray. Storage channels in disk arm optimization. In *2012 IEEE Symposium on Security and Privacy*, pages 52–52. IEEE Computer Society, 1991.

[21] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[22] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12. USENIX Association, 2012.

[23] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397, Aug. 1999.

[24] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – Crypto'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[25] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA*, pages 393–404. IEEE Computer Society, 2009.

[26] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using stopwatch. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:1–12, 2013.

[27] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11. ACM, 2011.

[28] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12. IEEE Computer Society, 2012.

[29] I. Molnar. Linux Kernel Documentation: CFS Scheduler Design. `http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[30] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, 2010.

[31] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[32] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[33] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *LNCS*, pages 200–210, Sept. 2001.

[34] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[35] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[36] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 194–199. IEEE, 2011.

[37] SPEC. SPECjbb2005 - Industry-standard server-side Java benchmark (J2SE 5.0). Standard Performance Evaluation Corporation, June 2005.

[38] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[39] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

[40] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based Defenses against Cross-VM Side-channels, 2014. Full version, available from authors' web pages.

[41] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen (short paper). In T. Ristenpart and C. Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, Oct. 2011.

[42] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[43] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07. ACM, 2007.

[44] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[45] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.

[46] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[47] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, CCS '13. ACM, 2013.

Preventing Cryptographic Key Leakage in Cloud Virtual Machines

Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, Huseyin Ulusoy
The University of Texas at Dallas
800 W. Campbell Rd, Richardson TX, 75080

Abstract

In a typical infrastructure-as-a-service cloud setting, different clients harness the cloud provider's services by executing virtual machines (VM). However, recent studies have shown that the cryptographic keys, the most crucial component in many of our daily used cryptographic protocols (e.g., SSL/TLS), can be extracted using cross-VM side-channel attacks. To defeat such a threat, this paper introduces HERMES, a new system that aims to protect the cryptographic keys in the cloud against any kind of cross-VM side-channel attacks by simply partitioning the cryptographic keys into random shares, and storing each share in a different VM. Moreover, it also periodically re-shares the cryptographic keys, thereby invalidating the potentially extracted partial ones. We have implemented HERMES as a library extension that is transparent to the application software, and performed deep case studies with a web and a mail server on Amazon EC2 cloud. Our experimental results show that the runtime overhead of the proposed system can be as low as 1%.

1 Introduction

Recent advances in cloud computing enable customers to outsource their computing tasks to the cloud service providers (CSPs). Typically, CSPs manage extensive amount of computational resources, and provide services, such as *Infrastructure-as-a-service* (IaaS) [40], *Platform-as-a-service* (PaaS) [31], *Software-as-a-service* (SaaS) [44]. By *outsourcing* core computing to the cloud, customers can mitigate the burden of resource management, and concentrate more on the core business tasks. A recent study on the cloud usage [3] reported that nearly 30% of enterprise IT organizations use public IaaS, such as Microsoft's Azure Service [12], Amazon's Elastic Compute Cloud (EC2) [4], or Google's Compute Engine [9].

Despite its numerous advantages, cloud computing also introduces new challenges and concerns, primarily the security and privacy risks [48]. The concerns simply stem from outsourcing critical data (e.g., health records, social security numbers, or even cryptographic keys) and/or computing capabilities to a distant computing environment, where the resources are shared with other potentially untrusted customers.

In particular, to increase efficiency and reduce costs, a CSP may place multiple *virtual machines* (VMs), belonging to different customers, to the same physical machine. In such an execution platform, VMs should be logically isolated from each other to protect the privacy of each client. The CSPs use *virtual machine monitors* (VMM) to realize logical isolation among VMs running on the same physical machine. However, recent studies show that a clever adversary can perform *cross-VM side-channel attacks* (for brevity, cross-VM attack) to learn private information that resides in another VM, even under carefully enforced logical isolation in public cloud infrastructures. More specifically, Ristenpart et al. [41] showed heuristics to improve an adversary's capabilities to place its VMs alongside the *victim* VMs, and learn crude information (e.g., aggregate cache usage). Most recently, Zhang et al. [51] managed to extract ElGamal decryption keys by cross-VM attacks. These studies have clearly demonstrated that logical isolation and trustworthy cloud provider are not necessarily enough to guarantee the security of sensitive information.

It would be too optimistic to assume that an adversary is only limited to the two aforementioned attacks. Unfortunately, there exists a wide variety of side-channel attacks, each with its own setup and methodology (e.g., [13–15, 19, 26, 28, 34, 43]). Simply, the absence of such attacks on public cloud infrastructures does not necessarily mean that they are inapplicable. In fact, there are side-channel attacks that target virtualized environments, and leverage timings of cryptographic operations or monitoring of common resource usage [39, 47]. Those attacks may be just one step behind being directly applicable to the public cloud setting; which is why proposing prevention mechanisms is extremely vital for the security and

privacy of the sensitive data in the VMs including the cryptographic keys.

To this end, we present HERMES, a system that remedies the cryptographic key disclosure vulnerabilities of VMs in the public cloud by using well-established cryptographic tools such as *Secret Sharing* and *Threshold Cryptography*. Specifically, the key technique in our system is to partition a cryptographic key into several pieces, which are computed using threshold cryptosystems, and to store each *share* on a different VM. This makes it harder for an adversary to capture the complete cryptographic key itself, since it now has to extract shares from multiple VMs (note that there is no single key or a centralized key anymore in HERMES). To further improve the resilience, the *same* cryptographic key is *re-shared* periodically, such that a share is *meaningful* in only one time period. Consequently, we introduce two significant challenges against a successful attack: (i) multiple VMs should be attacked, and (ii) each attack should succeed within a certain time period. As a proof-of-concept, we apply HERMES to protect the cryptographic keys of *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS) protocols.

Contributions. In short, this paper makes the following four contributions:

1. We present HERMES, a novel system to prevent the leakage of cryptographic keys in cloud VMs via mathematically proven techniques – *Secret Sharing* and *Threshold Cryptography*.

2. As a proof-of-concept, we build a prototype of HERMES and apply it to protect the SSL/TLS cryptographic keys, which is significantly more resilient to any cross-VM attack.

3. We empirically evaluate HERMES with micro benchmarks, and case studies for a web server and a mail server, and show that with optimal setup, HERMES can operate with overheads as low as 1%.

4. We formalize the problem of finding *good* HERMES configurations, which minimizes the security risk for given monetary and performance constraints.

Organization. The rest of the paper is structured as follows: We start by providing some background information in §2 about the protocols and techniques used in HERMES. It is followed by the threat model in §3, and the full technical details of HERMES in §4. Then, we evaluate HERMES regarding its efficiency in §5, and discuss its security in §6. Finally, we review the related work in §7, and conclude in §8.

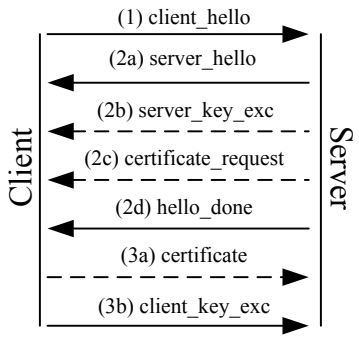


Figure 1: Overview of SSL Protocol Handshake.

2 Background

2.1 SSL/TLS Protocols

SSL and TLS are widely accepted communication protocols to establish a secure channel between two mutually-distrusting parties, where two protocols contain only a few minor differences [16, 25, 29, 37]. For brevity, we will refer the protocols as SSL; and any statement for SSL is also applicable to TLS.

The SSL protocol consists of a *handshake* and a *record* process, where the parties in the protocol are called the *client* and the *server*. In the handshake process, they use *public key cryptography* (PKC) to authenticate each other and agree on the session keys. The session keys are bound for only one session, and used for confidentiality and integrity. To calculate session keys, parties need to share a *master secret*, which is derived from random data exchanged, and *pre-master secret*.

Fig. 1 shows an overview of the handshake process. First, the client starts by sending *client_hello* message (**Step 1**), which contains a set of supported cryptographic algorithms (*cipher suites* in SSL terms), and some random data to be used in key generation. Then, the server sends its certificate, some random data, and the accepted cipher suite (**Step 2a**); and key exchange parameters if necessary (**Step 2b**). Moreover, if the server wants to authenticate the client, it requests the client's certificate (**Step 2c**). The server finishes by sending *hello_done* message (**Step 2d**). If requested, the client sends its certificate to the server, along with some random data signed by its private key (**Step 3a**). Next, it creates a random pre-master secret, encrypts it with the server's public key, and sends to the server (**Step 3b**). Now, both parties can calculate the master secret from the pre-master secret and random data using protocol specific combinations of pseudo-random functions.

Based on the chosen cipher suite, the number and the content of the messages may vary. For instance, when Diffie-Helman (DH) [22] is used for pre-master agree-

ment, the parties sign their DH parameters with their private keys, and send them in **Step 2b** and **Step 3b**. On the other hand, they may use RSA to agree on the pre-master secret, where the client encrypts the pre-master secret with the server's RSA public key, and the server decrypts it using its private key.

2.2 RSA Variants

The following variants of the RSA algorithm alter the way that a message is exponentiated with the private key. In both versions, the dealer holds a public-private RSA key pair, and wants to partition the private key over l non-colluding parties.

Distributed RSA (D-RSA). Given a private key d, D-RSA uses *additive secret sharing*, and partitions d into l random shares $d_1, \ldots, d_l$, where $d \equiv d_1 + \ldots + d_l \bmod \phi(n)$, n is the modulus, and ϕ is Euler's totient function. Given the public key (n, e) and a share, none of the parties can learn anything about d. Furthermore, an adversary should capture all l shares to learn d.

To exponentiate a message $M \in \mathbb{Z}_n$ with d, one of the parties acts as the *combiner*, whose job is to combine partial results from all parties. Each party p_i for $1 \leq i \leq l$ calculates M^{d_i}, and sends it to the combiner. Then, the combiner simply multiplies each message and finalizes the operation. At the end of the process, the combiner does not learn anything about the private key, but only the final result M^d. For a detailed security analysis, we refer to the original paper [24].

Threshold RSA (T-RSA). In this variant, the given private key is partitioned using *shamir secret sharing*, in which only $1 < k \leq l$ shares are needed to complete an exponentiation with d. The key technique in T-RSA is to embed the private key into a degree $(k-1)$ polynomial, evaluate the polynomial on l different points, and share the results over the set of parties. Once again, a party cannot learn the partitioned private key simply from the public key and its share.

To exponentiate a message, k parties are chosen uniformly at random, where the combiner once again does not learn anything other than M^d. On the other hand, an adversary should capture k shares to learn the private key. In App. B, we present more details on private key partitioning and usage, while an intensive security analysis is performed in the original paper [42].

3 Threat Model

Entities. The entities in our threat model are the *Cloud Service Provider* (CSP), the *Defender*, and the *Adversary*, where the last two are simply the clients of the first. The *CSP* offers IaaS and PaaS, which the clients can benefit by initiating VMs. The defender and the adversary use the same CSP, where the latter attacks the former to retrieve private information. Although the CSP has a potential to violate its clients' privacy and integrity, we assume that the CSP is trusted. This is a valid assumption, since (i) *Service Level Agreements* (SLA) provide a clear-cut distinction between what a CSP can and cannot perform on a client's data/VM, and (ii) disobeying a SLA may impose prohibiting punishment for the CSP.

Logical isolation. To improve utilization, the cloud provider may perform multiplexing. Hence, multiple VMs may run on the same physical machine, which means a VM of the adversary may run on the same physical machine with a VM of the defender; and they may share the same physical resources (e.g., CPU, memory, hard-drives, cache). On the other hand, we have no distinction on the VMM that the CSP uses, as long as it provides logical isolation between the VMs on the same physical machine. We assume that the adversary knows the software running on the defender VMs, but cannot leverage the memory vulnerabilities of those software to compromise (i.e., to take full control of) the VMs.

Adversary's goal. The defender has multiple VMs in the cloud, and each one may contain a set of private cryptographic information. This set of information includes temporary symmetric keys (e.g., AES key), or a share of a distributed private key (e.g., share of an RSA key) that is created by HERMES. An adversary's aim is to capture PKC keys, since capturing a session key is useful for only one session, while acquiring PKC keys grants full access. To fulfill its desire, the adversary is allowed to execute any cross-VM attack in its disposal to extract private information from each VM, where the attack itself is applicable to the cloud setup. For instance, in access-driven attacks, the adversary may need to co-reside its VMs with the defender VMs. In such a case, the adversary should achieve co-residency, and make the attack applicable in a typical cloud setup. Moreover, the attacks on the defender VMs are not necessarily executed in serial manner. Each separate adversary VM can employ the cross-VM attack in parallel, if the nature of the attack enables such setup.

Finally, since the adversary uses the same cloud as the defender, we assume that all channels may be eavesdropped by the adversary, starting from right after the bootstrapping of HERMES. Giving this capability to the adversary may seem like an overprovision. However, we take precautions to handle even the worst case scenario, in which the adversary, somehow by-passing CSP's security mechanisms, listens to the conversations between the defender VMs.

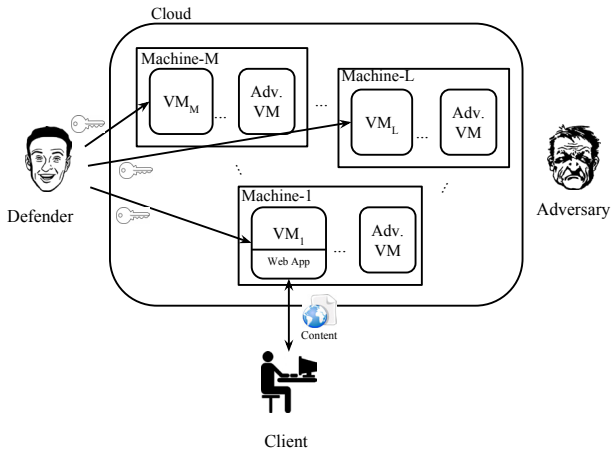


Figure 2: Overview of HERMES Layout.

Misc. We do not consider the placement of the defender VMs, and its effects on the security of HERMES. For instance, one platform (e.g., a region, a physical machine, etc.) may, somehow, be more susceptible to a certain range of cross-VM attacks; or one can claim that the more distributed the defender VMs, the better the security. Zhang et al. aim to physically isolate a defender VM as much as possible [50], thus preventing only access-driven side-channel attacks. Such precautions will tighten the defense against access-driven attacks; however, it will fail to stop the adversary from executing different attacks. On the other hand, HERMES aims to protect the cryptographic keys from all cross-VM attacks, no matter how the VMs are placed.

4 The Proposed System – HERMES

In SSL, a certificate may contain public parameters of different PKCs (e.g., RSA, DSA, ECC), which are employed to encrypt secret information, or to sign and show that certain temporary data is authentic. In HERMES, we assume that the parties use RSA as the PKC; however, extension to other PKCs is trivial as long as a threshold cryptosystem for that new PKC is provided.

Setup. Figure 2 shows an overview of the entities in HERMES: the defender, the adversary, l number of VMs that belong to the defender, and the clients who want to establish secure connection to the defender's VMs using SSL and benefit from the defender's web application. The defender holds a set of private RSA keys, and partitions them over the set of defender's VMs. Each VM holds one share for each partitioned private key, and they act together to exponentiate with it. The VM that directly talks with the client is called the *combiner*, while the remaining VMs are called *auxiliary VMs*. The adversary aims to learn at least one of the partitioned RSA private keys by (i) performing cross-VM attacks on each VM to capture its shares, and by (ii) listening each message flowing between the VMs. To achieve secure communication, each channel is established using our enhanced SSL protocol. More specifically, inter-VM channels are established with mutual verification (i.e., both end of the parties authenticate each other), while only the combiner VM is authenticated in a channel between that VM and a client/defender. The defender re-shares the same private keys every τ seconds. The time window between two consecutive re-sharing moments will be referred to as an *epoch*, while the shares of a private key in any two sessions are independent.

Modes. HERMES has two modes of operation, namely D-RSA and T-RSA modes, using the corresponding RSA variant (cf. §2.2). When the system runs in D-RSA mode, the adversary has to capture all shares of a private key to learn the key itself; whereas in T-RSA mode, it has to capture at least k shares. The benefits of the second mode are two-fold: (i) The system is more fault-tolerant to server failures, and (ii) the system can achieve better utilization by distributing work among different subsets of VMs, especially when $k \leq (l/2)$.

Stages. The execution of HERMES is composed of several stages: (i) Partitioning a private key (§4.2); (ii) Bootstrapping the system by handing in the initial set of shares, and establishing initial inter-VM SSL channels (§4.3); (iii) Establishing connection between a defender VM and a client (§4.4); (iv) Renegotiating an inter-VM SSL channel (§4.5); and (v) Distributing new shares of the same private keys (§4.6).

4.1 Enhancing the SSL Protocol

In SSL, the communicating parties may execute *mutual verification* or *server-only* verification. In any case, the server uses its private key at two possible steps (cf. Fig. 1): (i) After **Step 2a** to sign temporary parameters; (ii) after **Step 3b** to decrypt the pre-master secret. On the other hand, the client uses its private key before **Step 3a** only in the mutual verification. With respect to a regular SSL execution, we change the way that the server or client computes the modular exponentiation of a message with its RSA private key at those steps.

Fig. 3 shows the outline of our modifications in a server-only verified SSL execution. The client performs SSL handshake with the combiner, while the VMs communicate over already established secure channels. After **Step 2a**, the combiner may create temporary key parameters and sign them in collaboration with the auxiliary VMs. It sends a `help_sign` message to all auxiliary VMs in D-RSA mode (or up to k in T-RSA mode), where the

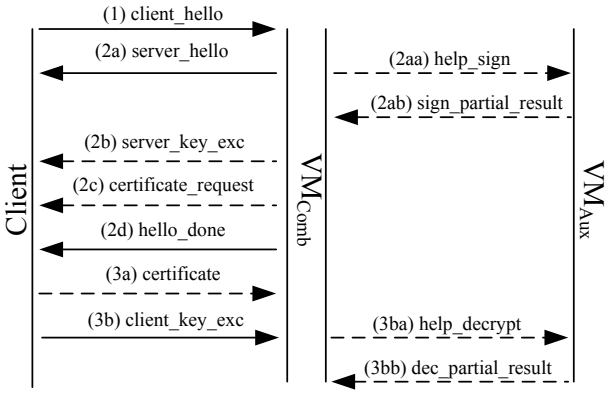


Figure 3: Security Enhanced SSL Outline for Server-only Verification.

message content is simply the parameters to be signed (**Step 2aa**). Each auxiliary VM in the computation calculates its partial result using its share of the private key and gives it to the combiner in the `sign_partial_result` message (**Step 2ab**). On the other hand, if the combiner has to decrypt an incoming message from the client, it sends a `help_decrypt` message to all auxiliary VMs in D-RSA mode (or up to k in T-RSA), containing the *masked* or plain version of the client's message (**Step 3ba**). Then, each auxiliary VM sends the computed partial result with the `dec_partial_result` message to the combiner (**Step 3bb**).

Whether or not the content of the `help_decrypt` message should be masked with a random number depends on the mode of operation. Even in the worst case, where the adversary knows each message exchanged between VMs, the combiner does not have to mask the message in D-RSA. The reason comes from the security of D-RSA. Assume the client sends $M^e \mod n$ to the combiner, where M is the pre-master secret, and (n,e) is the public key. In order to learn M, the adversary needs each VM's partial result, just like the combiner does. However, even if the adversary cracks down all secure channels and captures all messages, it can only learn $l-1$ parties' partial results, since the combiner does not send its partial result to anyone. Thus, the adversary cannot learn any useful information, and cannot compute M. If T-RSA is employed, then the combiner selects k VMs, $S = \{i_1, \ldots, i_k\}$, uniformly at random from the set of VMs and sends the message to them. There are two cases to consider: (i) If the combiner is included in S, then the message does not need masking similar to D-RSA case. The adversary needs k partial results, but can only capture $k-1$. (ii) Otherwise, the combiner masks the message with a random number; since the adversary may have captured k partial results sent from k different auxiliary VMs, and the other parameters in the calculation are public (e.g., $\Delta = l!$, a and b can be calculated from $gcd(e, 4\Delta^2)$), the adversary can now calculate M.

In addition to server-only verified SSL channels, HERMES necessitates mutual verification, since any two defender VMs, VM_i and VM_j, may perform key renegotiation to refresh session keys. Without loss of generality, assume that VM_i is the client in SSL protocol, while VM_j acts as the server. Now, both parties should communicate with the auxiliary VMs to perform operation with their own private keys. The server may need co-operation after steps **2a** and **3b**, while the client may need to sign random data with its private key before **Step 3a**. The server acts as mentioned in server-only authenticated Enhanced SSL. On the other hand, the client sends `help_sign` message to auxiliary VMs before **Step 3a**, and combines the partial results. By following those steps, two defender VMs can execute a successful handshaking process, using already established secure and authenticated SSL channels with the auxiliary VMs.

4.2 Partitioning Keys

Given an RSA key pair (n,e,d) and the number of VMs l, the defender performs partitioning and calculates the shares of each defender's VM. In case HERMES is running in T-RSA mode, the defender uses the third parameter k, minimum number of VMs needed to operate.

In D-RSA, the share of the i^{th} VM, denoted by sh_i, is simply a uniformly randomly chosen value from the domain $\mathbb{Z}_{\phi(n)}$, where $sh_1 + \ldots + sh_l$ is equal to d. Hence, the defender chooses $l-1$ random values, and calculates the final share as $sh_l = d - (sh_1 + \ldots + sh_{l-1}) \mod \phi(n)$.

On the contrary, key partitioning process is a bit more complicated in T-RSA. Algo. 1 shows an outline of preparations of each VM's share. For each subset $S_\alpha \subseteq \{1, \ldots, l\}$, where $|S_\alpha| = k$, the defender calculates the interpolation constants $\lambda_{0,j}^{S_\alpha}$, and exponents for each $VM_j \in S_\alpha$ (line 10-16). Moreover, the defender stores the modulus values for a, b in V_i's share for d, since i (the function input) states that the given private key belongs to VM_i (line 18-19).

4.3 Bootstrapping the System

The defender creates l VM instances on the CSP, and an RSA key pair (n_i, e_i, d_i) for each VM_i, $1 \le i \le l$. Next, she partitions each private key into shares and gives each VM a unique ID $i \in [1,l]$, the shares that correspond to that ID, and the certificate for the i^{th} RSA key pair.

At this stage, the VMs need to establish initial authenticated and secure SSL channels using our Enhanced SSL. However, as mentioned in §4.1, Enhanced SSL necessitates already established secure SSL channels to transfer messages between VMs. We have to make an assumption here, which will allow us to bypass this requirement, and

Algorithm 1 Preparing shares for T-RSA

Input: RSA Parameters n, $p = 2p' + 1$, $q = 2q' + 1$, e, d
Input: T-RSA parameters l, k, i
for $j \leftarrow 1$ to l **do**
    $sh_j \leftarrow \emptyset$
end for
$\Delta \leftarrow l!$
Calculate $S = \{S_1, \ldots, S_z\}$, where $z = \binom{l}{k}$, $\forall S_j \in S, |S_j| = k$, $S_j \subseteq \{1, \ldots, l\}$, and each S_j is distinct
$m = \phi(n)/4$
Create $f(X) = d + \sum_{j=1}^{k-1} a_j X^j$, where $\forall a_j \xleftarrow{R} \mathbb{Z}$
for all $S_\alpha \in S$ **do**
    for all $j \in S_\alpha$ **do**
        Calculate $\lambda_{0,j}^{S_\alpha}$
        $exp \leftarrow 4 \cdot \Delta \cdot f(j) \cdot \lambda_{0,j}^{S_\alpha} \bmod m$
        $sh_j \leftarrow sh_j \cup (i, S_\alpha, exp)$
    end for
end for
$(a, b) \leftarrow ecgd(e, 4\Delta^2)$, where $a4\Delta^2 + be = 1$
$sh_i \leftarrow sh_i \cup (a \bmod m, b \bmod m)$
return $sh_1, \ldots, sh_l$

to establish initial inter-VM SSL channels. We assume that the VMs, and the initial set of SSL channels are provisioned securely, i.e., no adversarial attack occurs until the initial set of SSL channels are established for inter-VM communications. This is a reasonable assumption, since (i) locating defender VMs on the cloud takes time [41], and (ii) the whole process of bootstrapping takes short time, especially if key-partitioning is performed beforehand. Once the initial inter-VM SSL channels are established, HERMES gets ready to serve the clients. Note that a defender VM uses the same RSA key pair for inter-VM and client connections.

Finally, in HERMES, we assume that the number of VMs is fixed throughout the entire life-time of execution. However, to augment HERMES capabilities with dynamic expansion of the system, one should care about the bootstrapping of those new VMs in terms of planting the initial secrets and initiating secure channels. As will be clear in §4.6, during the key re-sharing process, the defender may hand in secret shares to the newly added VMs. Still, introducing dynamic expansion via new VMs may lead to security vulnerabilities that should be investigated thoroughly.

4.4 Connecting to a Client

Once the bootstrapping stage is over, a client or the defender may request connection to a defender VM (i) to consume the services offered by the defender, or (ii) to distribute new shares for the private keys. In any case, the connection is established using server-only verified Enhanced SSL, where the connected VM takes the role of the combiner VM.

Assume the client wishes to connect to VM_i using Enhanced SSL. Throughout the handshaking process, VM_i interacts with the auxiliary VMs (i.e., all VMs other than VM_i), and performs distributed signing or decryption procedures as described in §4.1. The whole distributed operations are transparent to the client, while the combiner or any auxiliary VM learns nothing, but the result.

4.5 Inter-VM Key Renegotiation

Over time, any two defender VMs may decide to end one SSL session, and renegotiate keys for the next one. In such a case, those two VMs use their RSA key pairs, and perform a new handshaking process using our Enhanced SSL with mutual verification. Assume VM_i and VM_j decides to perform renegotiation, where VM_i and VM_j act as the client and server, respectively. Both VMs execute our Enhanced SSL handshaking process using already established SSL channels with the auxiliary VMs. When VM_i or VM_j needs to perform exponentiation with its private key, it collaborates with the auxiliary VMs, and calculates the result.

HERMES allows only one simultaneous key renegotiation at a given time, since an on-going process necessitates already established SSL channels. When two defender VMs start the process, it issues a warning to all VMs, blocking any other attempt for key renegotiation. Once the on-going procedure halts, HERMES removes the warning and allows the first renegotiation attempt.

4.6 Key Re-sharing

At the end of each epoch, the defender creates new shares for the same private RSA keys that were partitioned and distributed in the bootstrapping stage. In essence, it uses the key-partitioning algorithm discussed in §4.2 and generates shares that are independent from the previous ones. Then, it simply connects to each defender VM with our Enhanced SSL, as in §4.4, and hands in the new shares for all partitioned private keys.

The reason to adhere such a process is to mitigate the risk of private key disclosure, since the adversary may have already captured a set of shares for a partitioned private key. It is obvious that partitioning the same key for the second time will result in a different set of shares, which are totally independent from the previous. Hence, if the adversary did not capture enough shares to identify the exact key in one epoch, it will have to start from scratch, since those captured shares mean nothing in the next

		Setup									
		(1,1)	(2,2)	(3,3)	(4,4)	(5,5)	(6,6)	(7,7)	(8,8)	(9,9)	(10,10)
1 Cli.	Total	2.65	8.96	10.40	11.57	15.40	16.13	13.58	17.44	14.05	13.76
	Network	–	2.77	2.82	2.75	2.29	2.54	2.37	1.89	1.07	1.56
	Combine	–	1.78	1.76	1.82	2.25	2.18	1.87	1.99	2.00	1.93
10 Cli.	Total	5.54	31.74	37.85	45.87	43.59	40.82	48.65	52.77	50.64	52.82
	Network	–	19.42	21.67	14.29	18.92	19.89	25.69	21.68	18.11	14.09
	Combine	–	1.95	2.05	2.22	2.20	2.18	1.87	2.17	2.58	2.23
100 Cli.	Total	40.90	179.01	178.14	187.67	209.74	212.33	229.38	246.39	257.03	269.73
	Network	–	121.05	113.36	122.96	108.82	125.52	108.25	106.98	98.71	113.15
	Combine	–	2.16	2.14	2.01	2.10	2.09	2.03	2.11	2.81	2.28
1000 Cli.	Total	146.94	640.40	728.56	928.75	1023.75	904.59	989.32	1097.64	1001.06	1174.54
	Network	–	210.36	197.28	202.08	229.03	240.42	204.30	284.05	237.72	233.41
	Combine	–	2.26	2.08	1.96	2.18	2.17	2.20	2.43	2.24	2.62

Table 1: Average Connection, Network, and Combining Time Spent for D-RSA in milliseconds

epoch. The defender VMs do not immediately start using the new keys, since each defender should get the new shares, otherwise HERMES would have synchronization problems. Instead, a defender VM broadcasts a message to announce that it has the new shares. When all defender VMs have the new shares, they pass on to the next epoch, start using the new shares, and zeroise the old shares to leave no trace. Till then, the VMs continue using the *old* epoch shares.

5 Evaluation

We have implemented a prototype of HERMES atop the most commonly used open source SSL library, OpenSSL [10] v1.0.1e, the latest version as of this writing. Our implementation is a separate *shared library* compatible with the OpenSSL's *Engine API*. Without changing the OpenSSL source, programmers can plug-in our implementation and vary the way that RSA computations are performed with the private key. Meanwhile, we have also created multi-threaded applications (i) for the auxiliary VMs to establish SSL connections with the combiner VM, and to perform mathematical operations (e.g., exponentiation with the private key share); (ii) for the defender to partition the RSA private keys and hand in the shares to each defender VM. In this section, we present our evaluation result.

5.1 Experiment Setup

Case Studies. As it is challenging to exhaustively test HERMES with all the network benchmarks, we evaluated our system using a micro benchmark to profile the performance, and two representative case studies, in which SSL connection is necessary. The micro benchmark experiments evaluate the performance under varying system setups to target possible bottlenecks. Once the system dynamics are profiled, we execute two real-life case studies and check any efficiency deficits. The first case study is a web server, for which we used Apache HTTP Server [7] v2.4.4. A client connects to the server via HTTPS, and retrieves the default web page that comes with the application, which is a static HTML page of size $2KB$. The second case study is a mail server using Postfix v2.10 [11]. On top of that, we installed Dovecot [8] v2.2.4 as the IMAP(s)/POP3(s) server. A client connects to the Dovecot instance via IMAPS and checks the status of a mailbox, which contains a single mail of size $1KB$. Both server applications are executed with the *keep alive* property off (i.e., the server does not store SSL sessions, and performs a new handshake for every connection attempt by the clients).

One may argue that testing the web and mail servers with such low-sized content is applicable to real-world case. It is true that almost all web sites serve contents that may have much larger sizes. However, the purpose of the experiments is to put as much pressure as possible to HERMES in the given web and mail server case studies. As will be clear in the results, as the number of connections performed per unit time increases, HERMES acts more efficiently due to decreased network overhead. Hence, increasing the content sizes would increase the amount of time the server spends on processing a query, and decrease the number of requests per unit time. Instead, we used 1 and 2KB contents, and tried pushing HERMES as much as possible.

Benchmarks. To extract micro benchmark results, we developed applications that connect to the given defender VM, using given number of concurrent clients. For the web server application, we used two different benchmarking tools: Apache HTTP server benchmarking tool [5] (AB) v2.4.4, which allows us to send HTTPS queries with a variety of execution options; and Apache JMeter [6] (AJ) v2.9, where we used the default HTTPS request sampler that comes with the standard AJ binaries. For the mail server application, we used AJ again, with the default mail reader sampler. Similar to the server-side, we

		Setup								
		(10,2)	(10,3)	(10,4)	(10,5)	(10,6)	(10,7)	(10,8)	(10,9)	(10,10)
1 Clients	Total	12.06	12.27	13.44	11.57	16.10	17.94	16.80	19.80	13.76
	Network	4.97	5.14	4.89	2.75	5.07	5.59	5.29	1.15	1.56
	Combine	0.52	0.56	0.58	1.82	2.36	1.56	2.44	2.20	1.93
10 Clients	Total	19.78	23.22	28.14	45.87	39.48	48.99	49.97	60.70	52.82
	Network	9.72	10.15	10.19	14.29	16.30	23.15	27.17	34.03	14.09
	Combine	1.27	1.07	1.26	2.22	2.42	2.64	3.09	2.81	2.23
100 Clients	Total	54.90	71.07	88.31	187.67	130.17	163.24	182.12	206.00	269.73
	Network	11.77	25.27	37.77	122.96	69.74	84.05	82.96	121.32	113.15
	Combine	1.24	1.62	1.74	2.01	2.15	2.34	2.80	3.01	2.28
1000 Clients	Total	318.24	418.07	435.98	928.75	653.12	642.48	877.42	995.89	1174.54
	Network	88.12	123.72	130.21	202.08	196.29	212.05	214.20	216.97	233.41
	Combine	1.50	2.20	1.85	1.96	2.08	2.52	2.82	3.24	2.62

Table 2: Average Connection, Network, and Combining Time Spent for Fixed $l = 10$ in milliseconds

did not use the keep alive functionality in the benchmarking tools, which forces the clients to perform a new SSL handshaking for each request.

Hardware. For our experiments, we created 10 VM instances on Amazon EC2. The VM that serves that is augmented with the web and mail server applications is of the type *m1.xlarge* with 4 virtual CPU and 15GB of RAM. The remaining VMs are of the type *m1.small* with 1 virtual CPU, 1 virtual core, 1.7GB of RAM, and 64-bit Red Hat Enterprise Linux 6.4. The reason that we don't perform experiments with more number of VMs is that our results for 10 VMs are enough to extrapolate relations between HERMES modes, parameters (e.g., l, k, τ), and the performance metrics (e.g., average latency, throughput). All instances are created in the same EC2 region, US-West at Oregon. The instances communicate with each other over Amazon's private network, while a client or the defender interacts with the VMs over the public network. On the other hand, we used a single machine to send connection, web page, or mail check queries, where the machine is an IBM x3500m3 server with 16GB of RAM, and 4 quad-core CPUs at 2.4 GHz. Our client machine is located in our university campus, and is connected to the defender VMs over the Internet.

Parameters. We vary the number of concurrent clients from 1 to 1000 exponentially to observe the effects of increasing load on HERMES. We believe that the number 1000 is enough, since the number of web page views for most popular web sites goes up to 37 billion per year, which is approximately 1100 per second [1,2]. Each experiment ran for 5 minutes, and the average value of 5 runs is shown as the final result. As will be shown in the following subsection, we observe that a key re-sharing process takes approximately 50 *msec*. Combined with the observation that the average time to process a query may go up to 2 *sec*, we vary τ (i.e., the key re-sharing period) from 5 to 125 seconds.

We perform experiments using 10 VMs, and represent the setup as (l,k), where l is the number of active VMs, and k is the number of shares needed to calculate the RSA result. When l is equal to k, the system runs with D-RSA mode of operation using l VMs. Furthermore, $(1,1)$ represents the **single VM setup**, where the *default SSL (i.e., the one without our modification and there is only one key)* is used. Also, as l must be greater than or equal to k, it is important to note that we do not have any experiment set up of (l,k) where $l < k$.

5.2 Results

Micro Benchmarking. In this set of experiments, we aimed to observe the sole effects of HERMES on the performance, where the client simply connects to the combiner VM, and immediately closes the SSL channel, without sending any additional query. Naturally, we expected to observe a massive load on the combiner VM, since all it does is to establish SSL channel with the client using our enhanced SSL, and nothing else. Thus, the number of requests per unit time will be high, which will introduce an increased network overhead.

Table 1 shows the micro benchmark results for D-RSA with up to 10 VMs (e.g., $l = 10$). We vary the number of concurrent clients from 1 to 1000, and measure the average connection time, average time spent for inter-VM communication, and average time spent for combining partial results in milliseconds. It is observed that combining partial results from the auxiliary VMs do not incur more than 3 *msec* overhead; thus, does not affect a successful enhanced SSL connection in terms of efficiency. The reason is the simplicity of combining partial results (i.e., l modular multiplication). On the other hand, inter-VM communication dominates the overhead introduced by the enhanced SSL in D-RSA mode. Especially when the number of concurrent clients is 1000, average time required to execute an SSL connection exceeds 1 *sec* if $l > 7$. Thus, the network communication becomes the bot-

		Setup								
		(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	(10,2)
1 Clients	Total	8.96	12.03	11.23	11.97	12.37	13.47	12.16	9.58	12.06
	Network	2.77	4.77	4.82	5.28	5.10	4.77	5.42	4.76	4.97
	Combine	1.78	2.13	0.90	1.30	1.35	0.51	0.53	0.55	0.52
10 Clients	Total	31.74	33.45	23.77	25.34	23.57	20.18	18.81	19.26	19.78
	Network	19.42	19.10	11.00	9.78	9.62	8.20	10.49	9.26	9.72
	Combine	1.95	2.26	1.52	1.47	1.26	1.50	1.29	1.33	1.27
100 Clients	Total	179.01	164.65	95.30	82.50	80.98	66.95	73.26	58.08	54.90
	Network	121.05	95.84	52.52	38.03	30.00	25.90	25.93	25.26	11.77
	Combine	2.16	2.19	1.82	1.92	1.42	1.59	1.72	1.15	1.24
1000 Clients	Total	640.40	665.95	548.22	504.12	450.09	340.10	350.46	320.84	318.24
	Network	210.36	197.43	150.84	123.75	60.88	55.09	59.19	46.50	88.12
	Combine	2.26	1.93	1.91	1.91	1.55	1.41	1.42	2.36	1.50

Table 3: Average Connection, Network, and Combining Time Spent for Fixed $k = 2$ in milliseconds.

tleneck for D-RSA in high load, in case the combiner VM closes the connection right after a successful connection. In the results for our case studies, we observe that if the combiner has to process a request (e.g., prepare a web page, or check a mailbox) after a successful SSL connection, the network overhead decreases, which results in less average latency.

As previously mentioned, we introduce T-RSA mode to reduce the overhead by simply distributing work amongst different sets of VMs. Given the performance of 10 VMs in D-RSA mode, we check if the performance can be improved in T-RSA mode by reducing k (i.e., the number of needed VMs). Table 2 shows the results for a fixed $l = 10$ and varying number of k values. Furthermore, we perform experiments to observe the effect of increasing number of VMs for a fixed $k = 2$, and show the results in Table 3. The performance metrics and the client parameters are the same as in Table 1. It is observed that for a fixed l value, the average latency to complete an enhanced SSL connection drops down as k gets smaller, especially when $k \leq (l/2)$. The reason is that different sets of auxiliary VMs are consulted to complete a single SSL connection each time, which results in less network connection overhead. Hence, per each inter-VM connection, we observe less load, resulting up to 3 times better performance than D-RSA mode with same l value (e.g., between $(10,10)$ and $(10,2)$). On the other hand, it is still reasonable to pass to T-RSA, even if $k > (l/2)$, since decreasing k has, definitely, positive effects on the performance. For a fixed k value, increasing l by introducing new defender VMs has positive effects on the average time to complete an enhanced SSL connection, by simply reducing the inter-VM communication overhead. The value of k, indeed, affects the number of VMs that should be introduced to reduce the average completion time. We extrapolate that introducing nearly $2k$ new VMs into HERMES helps decreasing the overhead by nearly 50%. To solidify our derivations, we performed the same T-RSA experiments for different fixed values of k and l. For brevity, we moved the results of those experiments to App. A, from which the same observations can be easily made.

Key re-sharing time (*msec*)					
l	*Avg.Lat.*	l	*Avg.Lat.*	l	*Avg.Lat.*
2	17.59	5	37.89	8	54.64
3	23.60	6	43.65	9	57.34
4	29.51	7	49.82	10	61.28

Table 4: Average Completion Time for Key Re-sharing in milliseconds.

We, further, measured the average time of completion for a single key re-sharing process for varying number of defender VMs. Since, the number of connections that the defender has to do in the re-sharing process depends on l, but not on k, we performed the experiments in D-RSA mode with l number of VMs. Table 4 shows the results, where the defender re-shares the same partitioned keys every 5 seconds, and no other client attempts to connect to the VMs. The experiments ran for 5 minutes, and the average time to complete a single key re-sharing is calculated. We observe that the average time increases with the number of VMs, since the defender has to connect each VM separately, incurring additional inter-VM communication overhead. We see that the values in Table 4 coincides with the values in Table 1. When l is equal to 10, the defender has to make 10 simultaneous connections to HERMES, resulting a similar result as 10 clients D-RSA for the setup $(10,10)$. In case of high load (e.g., 1000 clients), the key re-sharing process would, definitely, take longer time. Thus, the optimal τ value for the key re-sharing epoch should be chosen while considering the server load, and the number of defender VMs.

Web Server. In our first case study, our aim is to show that the performance improves as the combiner executes a CPU intensive operation (e.g., prepare a web page) once connected to the client. The experimental setup is the same as the micro benchmarking setup, except now the combiner VM is a web server. When a client employs

SSL to connect to the combiner VM and retrieve a web page, the combiner VM collaborates with the auxiliary VMs, and executes our enhanced SSL.

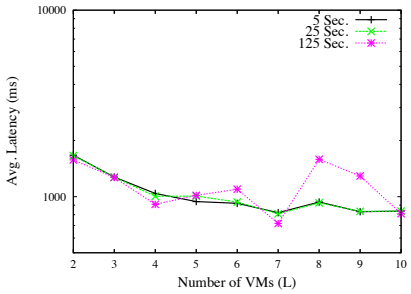
Fig. 4a and 4d show the results for HERMES in D-RSA mode, where the number of VMs changes from 1 to 10, and the number of concurrent clients changes between 1 to 1000 for τ of 125 *sec*. We use AB and AJ benchmarking tools, run the experiments for 5 minutes, and report the average time needed to execute a web page retrieval request, and the number of requests per second. We observe similar performance patterns for both of our metrics (e.g., performance decrease when l is increased) in compare to the micro benchmarks. However, the performance difference between the two end points (i.e., between $(1,1)$ and $(10,10)$) is narrower, due to more CPU-intensive processing done by the combiner. For 1000 concurrent clients, average latency and throughput in $(1,1)$ is 740 *msec* and 255 req/sec, respectively. On the other hand, the $(10,10)$ setup results in nearly 2 *sec* average latency, and 120 req/sec throughput. Compared to nearly 10 times increase in the micro benchmarking results, we see that the more CPU-intensive job the server does, the closer the gap between the $(1,1)$ and $(10,10)$ setups is.

Once again, we check if the performance can be boosted by passing to T-RSA mode, with decreased number of needed VMs. Fig. 4b and 4e show the results for fixed $l = 10$ and $\tau = 125$ *sec*. We observe that especially when $k \leq (l/2)$, the overhead reduces down to nearly 10% with respect to $(1,1)$ setup. For instance, in the $(10,5)$ setup, the average latency is 1088 ms, while the throughput is 220 req/sec. Even better, the throughput increases to 248 in $(10,3)$ setup, and to 250 in $(10,2)$ setup, which is just 2% less than $(1,1)$. The reason stems from distributing workload to more VMs by keeping seperate parts of the network busy at the same time, which reduces the inter-VM communication overhead.

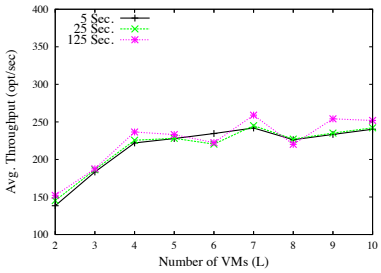
We remark that the results are gathered using the second *slowest* VM instances in Amazon EC2. The defender can instantiate stronger VM instances, with faster network, which will definitely improve the performance, since the network latency turns out to be the bottleneck. Furthermore, the defender can distribute the combiner role to multiple VMs to achieve further workload distribution.

The next results for the web server case study are given in Fig. 4c and 4f, where we measure the performance for varying l parameter and a fixed $k = 2$. We observe that having $l > 2k$ boosts the performance. Even in the $(4,2)$ setup, we measure that the average latency and throughput is 909 *msec* and 236 req/sec, respectively, which means less than 10% overhead for the second metric. When the number of VMs is more than $3k$, HERMES performs nearly the same as the $(1,1)$ setup.

To show that our choice of $\tau = 125$ *sec* does not have major effects on the overall performance, we vary the length of an epoch exponentially from 5 to 125 *sec* for different number of VMs, and fixed number of concurrent clients of 1000. We chose to execute epoch experiments for the fastest HERMES setup, namely fixed k with high l values, and to check if performance degradation occurs for decreased key re-sharing period. Fig. 5 shows the results for fixed $k = 2$ and 1000 concurrent clients, and varying τ values. We observe that even when $\tau = 5$ *sec*, the performance metrics behave similar to $\tau = 125$ *sec* case. This stems from the server being already loaded with enough concurrent clients, so that the seldom requests to re-share keys are only minor issues that does not take too much time to process.



(a) Avg. Latency for Epoch Times

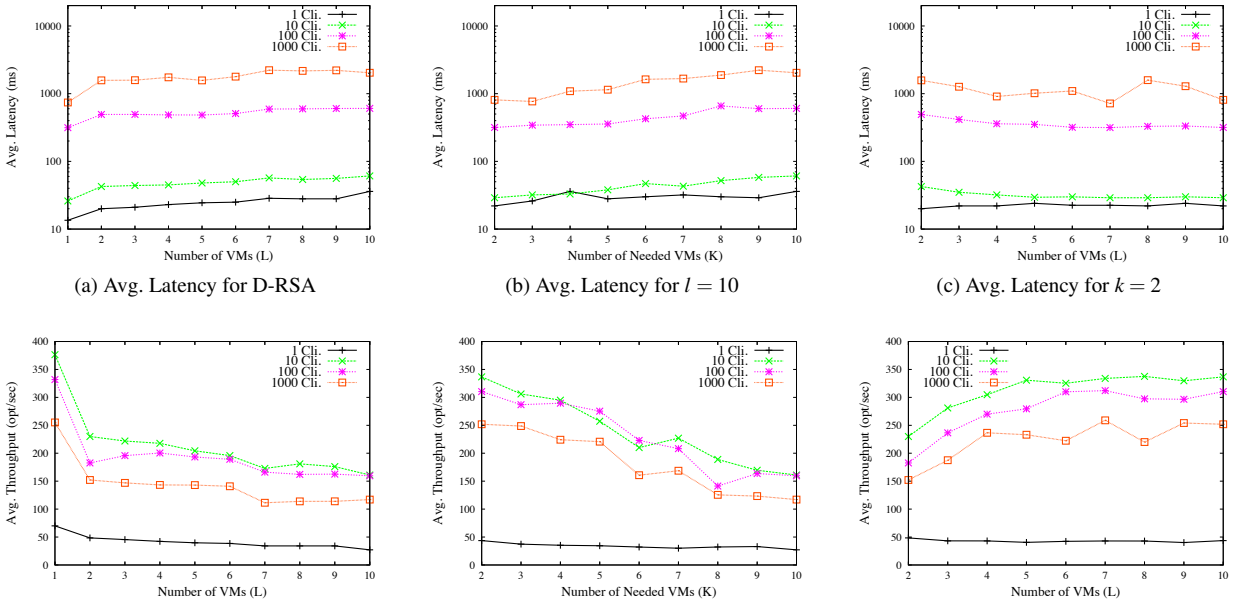


(b) Avg. Throughput for Epoch Times

Figure 5: Web Server results for $k = 2$ with varying τ

Mail Server. Mail Server is our second case study, where the clients establish connection using SSL via IMAPS protocol, and check a mailbox that contains a single mail. The default setting with regular SSL (i.e., $(1,1)$ setup) already results in an average 5758 *ms* latency, and 49.5 req/sec for 1000 concurrent clients. Hence, the combiner has to do more CPU-intensive operation for each client request. We claim that the margin between $(1,1)$ and $(10,10)$ setups will be less, since the network will be less occupied at a given time; thus, it will result in less inter-VM communication overhead.

Fig. 6 shows the results for the mail server case study, where the clients vary from 1 to 1000, and re-sharing period is 125 *sec*. First of all, we observe that the performance of each setup looks very similar, with nearly at most 8% overhead with respect to $(1,1)$. The reason to



(a) Avg. Latency for D-RSA (b) Avg. Latency for $l = 10$ (c) Avg. Latency for $k = 2$

(d) Avg. Throughput for D-RSA (e) Avg. Throughput for $l = 10$ (f) Avg. Throughput for $k = 2$

Figure 4: Web Server results

observe such a pattern is, as hypothesized, the fact that an average mail inquiry takes too much time to process. It causes low throughput values, resulting less number of SSL handshakes being made per unit time, which in turn causes less inter-VM communication overhead.

Once more, we observe that increasing the number of VMs for D-RSA mode has negative effects on the performance metrics, as shown in Fig. 6a and 6d. On the contrary, increasing l for fixed k value in T-RSA mode enhances the overall performance, due to better distribution of the defender VMs, as seen in Fig. 6c and 6f.

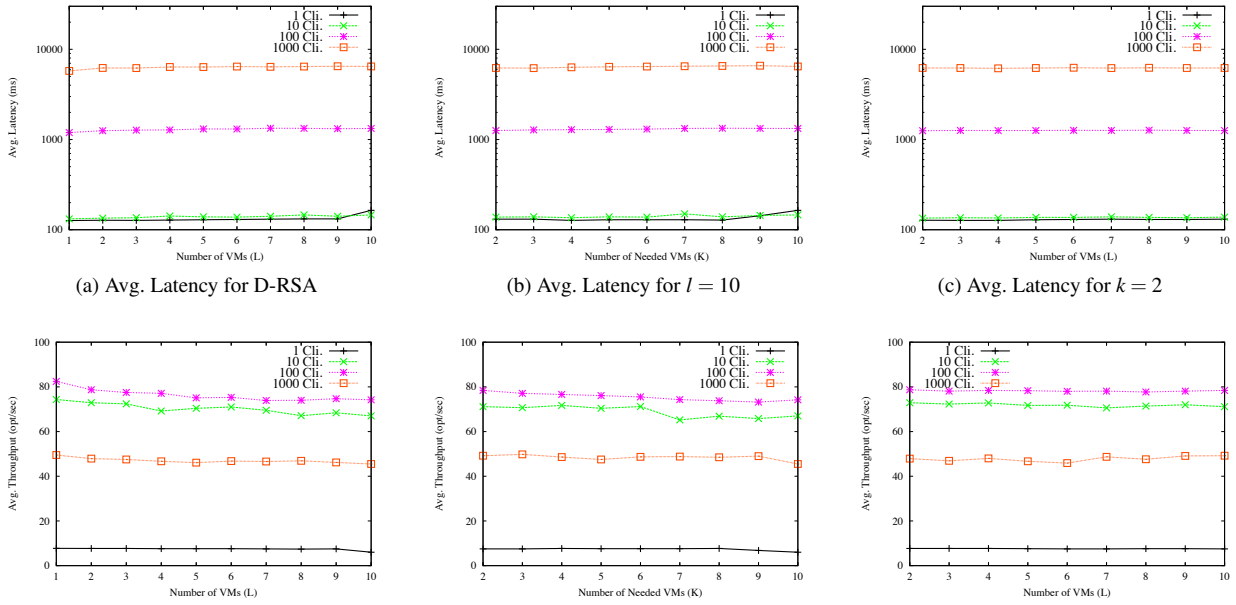
6 Security Analysis

The theoretical security of HERMES is based on the formally proven security of D-RSA and T-RSA, as discussed in §2.2. Combined with key re-sharing, the adversary should successfully capture at least l shares in D-RSA or k in T-RSA to calculate the shared cryptographic key. On the other hand, in practice, HERMES should give guarantees on the probability of a successful attack based on some assumptions on the nature of the attack and the system parameters (e.g., τ, l, k). The defender may have limited budget, or have certain performance requirements. In any case, HERMES must minimize any security risk by choosing l, k, and τ *optimally*. In this section, we first formalize the problem of finding such *optimal* values for those parameters, and then apply the optimization technique to a sample configuration: the micro benchmarking scenario discussed in §5. Our choice to apply optimization to only one configuration is due to space constraints; however, our approach is modular, and is easily applicable to any other cases.

6.1 Problem Formalization

In our formalization, we consider three main aspects: security, cost, and performance. Security aspect allows us to provide an upper bound on the possibility of a successful key extraction attack on HERMES for the given k, l, and τ values. Theoretically, increasing k and l, or decreasing τ will make it harder for the adversary to achieve its goal. However, increasing l implies more defender VMs running on the cloud, which increases the total cost. Moreover, our experiments showed that the performance degrades as l and k increase together. Hence, the optimal values should be assigned to k, l, τ for the given constraints (e.g., budget, performance limit).

Security Aspect. To quantify the probability of a successful attack in an epoch, we assume that the adversary has to start from scratch in each epoch, which implies that it loses all its previously acquired information. This is a valid assumption, since shares for each epoch are independent from one another, and a captured share does not contribute any information to the next epoch. The inability of conducting acquired information to the following epochs makes it convincing to model the probability of a successful attack as an *exponentially distributed* ran-



(a) Avg. Latency for D-RSA (b) Avg. Latency for $l = 10$ (c) Avg. Latency for $k = 2$

(d) Avg. Throughput for D-RSA (e) Avg. Throughput for $l = 10$ (f) Avg. Throughput for $k = 2$

Figure 6: Mail Server results

dom variable. Given the success rate parameter θ, the probability distribution for the attack is:

$$f(t) = \begin{cases} \frac{1}{\theta} e^{-t/\theta} & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Since the exponential distribution is *memoryless*[1] and the cryptographic key is re-shared in each epoch, we can simply assume that the input to f is the time difference from the last re-sharing moment. Then, given the length of the epoch τ, the probability of a successful attack is:

$$F(\tau, \theta) = \int_0^{\tau} f(t).dt = 1 - e^{-\tau/\theta} \quad (2)$$

Finally, assuming that the probability of capturing shares from a single VM is identical to and independent from all other VMs, the probability of capturing at least k shares from l defender VMs in an epoch is:

$$Sec(l, k, \tau, \theta) = \sum_{i=k}^{l} \binom{l}{i} (1 - e^{-\tau/\theta})^i (e^{-\tau/\theta})^{l-i} \quad (3)$$

Cost Aspect. Modeling monetary cost in HERMES is rather simple compared to the other two aspects. Assuming that the cloud provider does not charge money for the inter-VM communications, the total monetary cost is $Cost(l) = l.\beta$, where β is the unit cost of running a single VM on the cloud provider. The cost of communication with the client is also neglected, since this is not an additional cost incurred by HERMES.

Performance Aspect. The method to formalize the expected performance depends heavily on the application that HERMES is running for, and the metrics that the defender considers. For instance, one may value throughput more than the latency while running HERMES. On the other hand, the effects of changing parameters (i.e., k, l) in the mail server case study is far different than changing the same parameters in the micro benchmarking experiments. For brevity, we show the performance of HERMES for the given k and l as $Perf(l,k)$, and leave it to the defender to define the characteristics of the function.

Optimization Problem. Given the success rate parameter θ, the unit cost of a VM β, the budget limit L_{cost}, and the performance limit L_{perf}, the aim of the optimization problem is to minimize the probability of a successful attack in an epoch while keeping the total monetary cost below L_{cost} and the performance below L_{perf}. Formally, the optimization problem is:

$$\begin{aligned} \text{minimize:} \quad & Sec(l, k, \tau, \theta) \\ \text{subject to:} \quad & Cost(l) \leq L_{cost}, Perf(l,k) \leq L_{perf} \\ & l \geq k > 1, \tau > 0 \end{aligned}$$

6.2 Application to Micro Benchmarking

Modeling performance is highly dependent on the case study and the aimed configuration, thus it is challenging to apply the optimization to every single case. Instead, we targeted to optimize HERMES for 100 concurrent clients

	$\theta = 600$		$\theta = 3600$	
L_{cost}/yr	Conf.	Sec()	Conf.	Sec()
\$1820	$(2,2)$	$6.8 \cdot 10^{-5}$	$(2,2)$	$1.9 \cdot 10^{-6}$
\$3640	$(4,3)$	$2.2 \cdot 10^{-6}$	$(4,3)$	$3.7 \cdot 10^{-8}$
\$7280	$(8,5)$	$2.1 \cdot 10^{-9}$	$(8,5)$	$2.8 \cdot 10^{-13}$
\$14560	$(16,10)$	$1.1 \cdot 10^{-17}$	$(16,10)$	$2.1 \cdot 10^{-25}$

Table 5: Optimal setup and resulting successful attack probabilities in an epoch for fixed expected latency limit $L_{perf} = 150$ *msec*, and $\theta = \{600, 3600\}$

	$\theta = 600$		$\theta = 3600$	
L_{perf}	Conf.	Sec()	Conf.	Sec()
50 *msec*	$(16,6)$	$2.4 \cdot 10^{-9}$	$(16,6)$	$5.6 \cdot 10^{-14}$
100 *msec*	$(16,8)$	$2.7 \cdot 10^{-13}$	$(16,8)$	$1.7 \cdot 10^{-19}$
150 *msec*	$(16,10)$	$1.1 \cdot 10^{-17}$	$(16,10)$	$2.1 \cdot 10^{-25}$
200 *msec*	$(16,11)$	$5.4 \cdot 10^{-20}$	$(15,11)$	$4.9 \cdot 10^{-29}$

Table 6: Optimal setup and resulting successful attack probabilities in an epoch for fixed monetary budget $L_{cost} = \$14560/yr$, and $\theta = \{600, 3600\}$

in the micro benchmarking scenario, since all experiment results for the chosen configuration are given in §5.2 and App. B. For brevity, we make a further assignment of parameters by choosing re-sharing period as $\tau = 5$ *sec* and success rate parameter as $\theta = 3600$. $\tau = 5$ *sec* is the smallest value that we have tested, and is a valid value that allows HERMES to complete several computations in each epoch. Furthermore, choosing small re-sharing period will tighten the overall security, since the adversary has to complete the attack in a very short period. On the other hand, choosing θ as 3600 is due to the existing cross-VM attacks (i.e., [41, 51]), which necessitates hours to capture the cryptographic key. In an exponential distribution, expected *waiting time* to observe one success is θ. Since, we expect the attack to succeed in an hour, we assign $\theta = 3600$, representing the number of seconds in an hour. In addition, we check $\theta = 600$ to observe changes in optimal values.

In this example, we picked latency as the target performance metric to consider, assuming that the defender aimed to serve 100 concurrent clients as fast as possible. The important step to model performance is to figure out $Perf(l,k)$. To overcome this, we applied *multiple linear regression* on our experiment results, and came up with a formula that gives the *expected* latency value for the given l and k values. As it is challenging to test every possible formula, and increasing the number of variables may over-fit the training data, we chose a simple polynomial $Perf(l,k) = c_0 + c_1.l + c_2.k + c_3.(l/k)$ to model the expected latency, where the coefficients are $c_0 = 118$, $c_1 = -18$, $c_2 = 31$, and $c_3 = 7$. Finally, to observe the effects of different performance limits L_{perf}, we calculated optimal HERMES setups for $L_{perf} \in [50, 200]$. Finally, assuming that the defender will use the second cheapest VM instance on Amazon EC2, she will pay $\$0.104/hour$, which is approximately $\$910/yr$. We vary the monetary budget between $\$1820/yr$ and $\$14560/yr$ to check optimal values, which is simply $l \in [2, 16]$.

Table 5 shows the results of the optimization procedure for varying monetary budget, and fixed $L_{perf} = 150$. The results include the optimal HERMES setup and the probability of a successful attack in one epoch, for both $\theta = 3600$ and 600. We observe that as we increase the monetary budget, HERMES is allowed to run with more VMs, resulting in lower probabilities of success for the adversary. For instance, when the budget is $\$7280/yr$ and $\theta = 3600$, HERMES can be configured to run in $(8,5)$ setup, while the adversary has only $2.8 \cdot 10^{-13}$ chance to capture the partitioned cryptographic key.

Table 6 shows similar set of results, this time for fixed monetary budget of $\$14560/yr$, but varying expected performance limit L_{perf}. We deduce that as HERMES is allowed to respond slower, it can be configured to run with increased k, which decreases the attack success probability. For instance, increasing expected latency from 50 *msec* to 150 *msec* decreases the attack success probability nearly 8 and 11 fold, when θ is 600 and 3600, respectively.

Note that the probabilities are calculated for a successful attack in **one epoch**, one would question if the adversary would accomplish its goal in a longer period, say a year. For $\$7280/yr$ and an expected latency of 150 *msec* in our micro benchmarking case study, the probability of capturing the complete cryptographic key in one year is $1.8 \cdot 10^{-6}$ if the average time to capture a key is predicted as 3600 *sec*, which is a very small probability.

7 Related Work

Attacks. There exists a myriad of side-channel attacks with different assumptions and setups. The adversary may leverage observations made on the shared hardware to execute *access driven* attacks (e.g., [13, 14, 28, 43]); measure timings of certain cryptographic operations of the defender to perform *time-driven* attacks (e.g., [15, 19, 20, 34, 47]); or physically observe the defender machines and run *trace driven* attacks (e.g., [18, 26, 33]). The specific type of attacks, which HERMES aims to mitigate, is *cross-VM* side-channel attacks, in which both the defender and the adversary are customers to a third-party cloud infrastructure. Both entities initialize VMs in the cloud, where the victim's VMs are attacked by the adversary's VMs. Ristenpart et al. [41] showed the first cross-VM side-channel, in which they first *co-reside* their VMs with the defender VMs, and execute an access-driven attack to retrieve crude information (e.g., aggregate

cache usage). In another work, though not for adversarial purposes, Zhang et al. [50] present *HomeAlone* that performs a co-residency check between two VMs using classifiers on cache timing. In another attack, Zhang et al. push it one step further, and extract an ElGamal private key using cross-VM attacks [51]. Those works showed that VMs in public cloud infrastructures are vulnerable to side-channel attacks, and protection mechanisms are needed to secure private information.

Prevention. Among the variety of side-channel prevention techniques, the most popular ones are randomization-based approaches. *MIST* is one of such examples, in which the *square-and-multiply* method is extended with an additional division by a randomly chosen number [38, 45]. Other approaches include adding random noise between squaring and multiplying operations, or applying *always-multiply* techniques. To countermeasure those side-channel prevention techniques, Karlof et al. promotes *Hidden Markov Model* based cryptanalysis as a powerful tool [30]. On the other hand, Witteman et al. shows a trace driven side-channel attack to break down *always-multiply* technique and message binding in RSA [49]. Although the latter is trace driven, those two works show that even randomization based side-channel prevention approaches could have vulnerabilities that can be used by different types of adversaries.

There exist several works that aim to prevent side-channel attacks in public clouds. *HomeAlone* [50] uses co-residency checks to see if a VM is physically isolated from any other VM, and to achieve maximum physical isolation. Our work aims to prevent the leakage of private keys even if the adversary co-resides with the defender, whereas they aim to prevent access-driven side-channel attacks by assuring physical isolation. In *HyperSafe*, Wang and Jiang aim to provide hypervisor integrity throughout the execution [46]. We assume that the cloud provider and its infrastructure (including the hypervisor) are trusted. Other prevention mechanisms include [17], which aims to prevent side-channel attacks that use communication traffic; *StealthMem* that hides memory access patterns to protect private information [32]. Compared to these works, HERMES is applicable to any type of cross-VM attacks against cryptographic keys.

8 Conclusion

In this paper, we present HERMES, a novel system to protect cryptographic keys in cloud VMs. The key idea is to periodically partition a cryptographic key using additive or Shamir secret sharing. With two different case studies, we show that the overhead can be as low as 1%. With such small overhead in an average request, cryptographic keys become more leakage-resilient against any adversary. Furthermore, we model the problem of finding *optimal* parameters for the given monetary and performance constraints, which minimizes the security risk. Using our formal model, the defender can calculate the probability of a successful attack, and take precautions (e.g., increase the number of VMs, decrease epoch length). As a proof-of-concept, the current implementation of HERMES mainly focuses on the protection of the RSA private key, which is widely used in many daily web site and mail server communications. However, there exists a myriad of works on threshold signature schemes for different cryptosystems, (e.g., [21, 23, 27, 35, 36]), which may be applicable to HERMES with slight modifications.

9 Acknowledgement

We thank the anonymous reviewers for their insightful comments. This work was partially supported by Air Force Office of Scientific Research FA9550-12-1-0082 and FA9550-14-1-0119, National Institutes of Health Grants 1R0-1LM009989 and 1R01HG006844, National Science Foundation (NSF) Grants Career-CNS-0845803, CNS-0964350, CNS-1016343, CNS-1111529, CNS-1228198 and Army Research Office Grant W911NF-12-1-0558.

References

[1] Us web statistics released for may 2012: which sites dominate, and where do we go for online news? http://www.theguardian.com/news/datablog/2012/jun/22/website-visitor-statistics-nielsen-may-2012-google, 2012.

[2] Internet 2012 in numbers. http://royal.pingdom.com/2013/01/16/internet-2012-in-numbers/, 2013.

[3] Public, private and hybrid clouds when, why and how they are really used. Tech. rep., Summary report, Neovise, 2013.

[4] Amazon elastic compute cloud. http://aws.amazon.com/pricing/ec2/, 2014.

[5] Apache http server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html, 2014.

[6] The apache jmeter desktop application. http://jmeter.apache.org/, 2014.

[7] Apache: The number one http server on the internet. http://httpd.apache.org/, 2014.

[8] Dovecot, an open source imap and pop3 email server. http://www.dovecot.org, 2014.

[9] Google compute engine. https://cloud.google.com/products/compute-engine, 2014.

[10] The openssl project. http://www.openssl.org, 2014.

[11] The postfix home page. http://www.postfix.org/start.html, 2014.

[12] Windows azure. http://www.windowsazure.com/en-us/, 2014.

[13] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer, 2010, pp. 110–124.

[14] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security* (2007), ACM, pp. 312–320.

[15] ACIIÇMEZ, O., SCHINDLER, W., AND KOÇ, Ç. K. Cache based remote timing attack on the aes. In *Topics in Cryptology–CT-RSA 2007*. Springer, 2006, pp. 271–286.

[16] ALLEN, C., AND DIERKS, T. The tls protocol version 1.0.

[17] BACKES, M., DOYCHEV, G., AND KOPF, B. Preventing side-channel leaks in web traffic: A formal approach. In *NDSS* (2013).

[18] BERTONI, G., ZACCARIA, V., BREVEGLIERI, L., MONCHIERO, M., AND PALERMO, G. Aes power attack based on induced cache miss and countermeasure. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on* (2005), vol. 1, IEEE, pp. 586–591.

[19] BRUMLEY, B. B., AND TUVERI, N. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*. Springer, 2011, pp. 355–371.

[20] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks 48*, 5 (2005), 701–716.

[21] DESMEDT, Y. Some recent research aspects of threshold cryptography. In *Information Security*. Springer, 1998, pp. 158–173.

[22] DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *Information Theory, IEEE Transactions on 22*, 6 (1976), 644–654.

[23] FOUQUE, P.-A., AND POINTCHEVAL, D. Threshold cryptosystems secure against chosen-ciphertext attacks. In *Advances in CryptologyASIACRYPT 2001*. Springer, 2001, pp. 351–368.

[24] FRANKEL, Y. A practical protocol for large group oriented networks. In *Advances in Cryptology EUROCRYPT 1989* (1990), Springer, pp. 56–61.

[25] FREIER, A. The ssl protocol version 3.0. http://ci.nii.ac.jp/naid/10015295976/en/, 1996.

[26] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems CHES 2001* (2001), Springer, pp. 251–261.

[27] GENNARO, R., JARECKI, S., KRAWCZYK, H., AND RABIN, T. Robust threshold dss signatures. In *Advances in CryptologyEUROCRYPT96* (1996), Springer, pp. 354–371.

[28] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 490–505.

[29] HICKMAN, K., AND ELGAMAL, T. The ssl protocol. *Netscape Communications Corp 501* (1995).

[30] KARLOF, C., AND WAGNER, D. Hidden markov model cryptanalysis. In *Cryptographic Hardware and Embedded Systems–CHES 2003: 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings* (2003), vol. 5, Springer, p. 17.

[31] KELLER, E., AND REXFORD, J. The platform as a service model for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (2010), vol. 4, USENIX Association.

[32] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 11–11.

[33] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis. In *Advances in Cryptology CRYPTO 1999* (1999), Springer, pp. 388–397.

[34] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology CRYPTO 1996* (1996), Springer, pp. 104–113.

[35] KUROSAWA, K. New eigamal type threshold digital signature scheme. *IEICE transactions on fundamentals of electronics, communications and computer sciences 79*, 1 (1996), 86–93.

[36] LANGFORD, S. K. Threshold dss signatures without a trusted party. In *Advances in CryptologyCRYPT095*. Springer, 1995, pp. 397–409.

[37] OPPLIGER, R. *SSL and TLS: Theory and Practice*. Artech House, 2009.

[38] OSWALD, E., AND AIGNER, M. Randomized addition-subtraction chains as a countermeasure against power attacks. In *Cryptographic Hardware and Embedded SystemsCHES 2001* (2001), Springer, pp. 39–50.

[39] OWENS, R., AND WANG, W. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International* (2011), IEEE, pp. 1–8.

[40] PRODAN, R., AND OSTERMANN, S. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In *Grid Computing, 2009 10th IEEE/ACM International Conference on* (2009), IEEE, pp. 17–25.

[41] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.

[42] SHOUP, V. Practical threshold signatures. In *Advances in Cryptology EUROCRYPT 2000* (2000), Springer, pp. 207–220.

[43] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[44] VAQUERO, L. M., RODERO-MERINO, L., CACERES, J., AND LINDNER, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review 39*, 1 (2008), 50–55.

[45] WALTER, C. D. Mist: An efficient, randomized exponentiation algorithm for resisting power analysis. In *Topics in Cryptology CT-RSA 2002*. Springer, 2002, pp. 53–66.

[46] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 380–395.

[47] WEISS, M., HEINZ, B., AND STUMPF, F. A cache timing attack on aes in virtualization environments. In *Financial Cryptography and Data Security*. Springer, 2012, pp. 314–328.

[48] WINKLER, V. Cloud computing: Cloud security concerns. Tech. rep., 2011.

[49] WITTEMAN, M. F., VAN WOUDENBERG, J. G., AND MENARINI, F. Defeating rsa multiply-always and message blinding countermeasures. In *Topics in Cryptology–CT-RSA 2011*. Springer, 2011, pp. 77–88.

[50] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 313–328.

[51] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 305–316.

		Setup							
		(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)
1 Clients	Total	9.58	10.70	11.53	15.88	13.93	13.97	14.13	14.05
	Network	4.76	4.49	4.85	5.30	5.01	4.84	2.45	1.07
	Combine	0.55	1.44	1.18	1.65	1.77	1.65	2.75	2.00
10 Clients	Total	19.26	21.29	24.14	31.43	32.81	39.67	46.73	50.64
	Network	9.26	10.84	12.87	15.85	14.55	22.92	23.74	18.11
	Combine	1.33	1.51	1.68	2.07	2.61	2.83	2.76	2.58
100 Clients	Total	58.08	68.32	91.87	144.08	164.54	209.26	247.54	257.03
	Network	25.26	37.91	51.19	98.69	108.02	101.33	111.37	98.71
	Combine	1.15	1.64	2.01	2.24	2.56	2.14	2.81	2.81

Table 7: Average Connection, Network, and Combining Time Spent for Fixed $l = 9$ in milliseconds

		Setup							
		(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)
1 Clients	Total	10.4	12.98	12.04	11.96	13.33	11.57	10.70	12.27
	Network	2.82	6.57	5.11	5.04	5.28	4.92	4.49	5.14
	Combine	1.76	1.46	1.66	1.51	1.90	1.49	1.44	0.56
10 Clients	Total	37.85	35.85	29.35	24.22	24.31	21.66	21.29	23.22
	Network	21.67	21.81	15.76	12.43	11.53	10.56	10.84	10.15
	Combine	2.05	2.02	1.97	1.69	1.71	1.57	1.51	1.07
100 Clients	Total	178.14	209.99	146.54	99.47	86.62	79.72	68.32	71.07
	Network	113.36	158.35	112.46	67.47	61.25	51.57	37.91	25.27
	Combine	2.14	2.49	2.00	1.85	1.90	1.65	1.64	1.62

Table 8: Average Connection, Network, and Combining Time Spent for Fixed $k = 3$ in milliseconds

A Additional Experiments

Tables 7 and 8 show the results for the micro benchmark.

B T-RSA Details

Key partitioning: The dealer creates two strong primes $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are also prime numbers. Next, it creates a random prime number $e > l$, and calculates $d = e^{-1} \ mod \ m$, where $m = p'q'$.

Then, the dealer creates a random polynomial $f(X) = \sum_{i=0}^{k-1} a_i X^i \in \mathbb{Z}[X]$, where $a_0 = d$, and $a_1, \ldots, a_{k-1}$ are random integers in $\mathbb{Z}$. Next, the dealer computes each party p_i's share as $s_i = f(i) \ mod \ m$. The public key is (n, e), while each party is given s_i as their share of the private key.

Using the secret key: For a given message $M \in \mathbb{Z}_n^*$, the chosen combiner selects a subset of the parties, $S = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, l\}$, where $|S| = k$, and sends M to each party in S. Each selected party p_{i_j} performs the following set of operations:

1. $\Delta = l!$
2. $\lambda_{0,i_j}^S = \Delta \frac{\prod_{i_x \in S \setminus \{i_j\}} -i_x}{\prod_{i_x \in S \setminus \{i_j\}} (i_j - i_x)}$
3. $w_{i_j} = M^{4\Delta s_{i_j} \lambda_{0,i_j}}$

λ_{0,i_j}^S is the polynomial interpolation constant for p_{i_j} in set S, where $\Delta f(0) \equiv \sum_{i_j \in S} \lambda_{0,i_j}^S f(i_j) \ mod \ m$. Once the combiner gets a partial result, w_{i_j}, from each party in S, it computes $w = \prod_{i_j \in S} w_{i_j}$. Then, it executes the extended Euclidean algorithm for e and $e' = 4\Delta^2$, and gets integers a and b, where $e'a + eb = gcd(e', e) = 1$. The greatest common divisor of e and e' is 1, since e is a prime number, and each factor of e' is smaller than e. Finally, the combiner computes $y = w^a M^b$ as the final result.

The final value y is in fact $M^d \ mod \ n$:

$$w \equiv \prod_{i_j \in S} w_{i_j} \equiv \prod_{i_j \in S} M^{4\Delta s_{i_j} \lambda_{0,i_j}} \equiv M^{4\Delta^2 d} \equiv M^{e'd} \ mod \ n$$

$$y \equiv w^a M^b \equiv M^{ae'd+b} \equiv M^{d(1-eb)+b} \equiv M^d \ mod \ n$$

Notes

[1] In a memoryless probability distribution, the cumulative probability depends on the distance from the starting time of the distribution to the current time.

FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

Yuval Yarom Katrina Falkner
The University of Adelaide

Abstract

Sharing memory pages between non-trusting processes is a common method of reducing the memory footprint of multi-tenanted systems. In this paper we demonstrate that, due to a weakness in the Intel X86 processors, page sharing exposes processes to information leaks. We present FLUSH+RELOAD, a cache side-channel attack technique that exploits this weakness to monitor access to memory lines in shared pages. Unlike previous cache side-channel attacks, FLUSH+RELOAD targets the Last-Level Cache (i.e. L3 on processors with three cache levels). Consequently, the attack program and the victim do not need to share the execution core.

We demonstrate the efficacy of the FLUSH+RELOAD attack by using it to extract the private encryption keys from a victim program running GnuPG 1.4.13. We tested the attack both between two unrelated processes in a single operating system and between processes running in separate virtual machines. On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round.

1 Introduction

To reduce the memory footprint of a system, the system software shares identical memory pages between processes running on the system. Such sharing can be based on the source of the page, as is the case in shared libraries [13, 26, 42]. Alternatively, the sharing can be based on actively searching and coalescing identical contents [6, 55]. To maintain the isolation between non-trusting processes, the system relies on hardware mechanisms that enforce read only or copy-on-write [13, 40] semantics for shared pages. While the processor ensures that processes cannot change the contents of shared memory pages, it sometimes fails to block other forms of inter-process interference.

One form of interference through shared pages results from the shared use of the processor cache. When a process accesses a shared page in memory, the contents of the accessed memory location is cached. Gullasch et al. [29] describes a side channel attack technique that utilises this cache behaviour to extract information on access to shared memory pages. The technique uses the processor's `clflush` instruction to evict the monitored memory locations from the cache, and then tests whether the data in these locations is back in the cache after allowing the victim program to execute a small number of instructions.

We observe that the `clflush` instruction evicts the memory line from all the cache levels, including from the shared Last-Level-Cache (LLC). Based on this observation we design the FLUSH+RELOAD attack—an extension of the Gullasch et al. attack. Unlike the original attack, FLUSH+RELOAD is a cross-core attack, allowing the spy and the victim to execute in parallel on different execution cores. FLUSH+RELOAD further extends the Gullasch et al. attack by adapting it to a virtualised environment, allowing cross-VM attacks.

Two properties of the FLUSH+RELOAD attack make it more powerful, and hence more dangerous, than prior micro-architectural side-channel attacks. The first is that the attack identifies access to specific memory lines, whereas most prior attacks identify access to larger classes of locations, such as specific cache sets. Consequently, FLUSH+RELOAD has a high fidelity, does not suffer from false positives and does not require additional processing for detecting access. While the Gullasch et al. attack also identifies access to specific memory lines, the attack frequently interrupts the victim process and as a result also suffers from false positives.

The second advantage of the FLUSH+RELOAD attack is that it focuses on the LLC, which is the cache level furthest from the processors cores (i.e., L2 in processors with two cache levels and L3 in processors with three). The LLC is shared by multiple cores on the same processor die. While some prior attacks do use the

LLC [47, 60], all of these attacks have a very low resolution and cannot, therefore, attain the fine granularity required, for example, for cryptanalysis.

To demonstrate the power of FLUSH+RELOAD we use it to mount an attack on the RSA [48] implementation of GnuPG [27]. We test the attack in two different scenarios. In the same-OS scenario both the spy and the victim execute as processes in the same operating system. In the cross-VM scenario, the spy and the victim execute in separate, co-located virtual machines. Both scenarios were tested in a local lab settings on otherwise idle machines.

By observing a single signing or decryption round, the attack extracts 98.7% of the bits on average in the same-OS scenario and 96.7% in the cross-VM scenario, with a worst case of 95% and 90%, respectively.

The rest of this paper is organised as follows. The next section presents background information on page sharing, cache architecture and the RSA encryption. Section 3 describes the FLUSH+RELOAD technique, followed by a description of our attack on GnuPG in Section 4. Mitigation techniques are presented in Section 5, and the related work in Section 6.

2 Preliminaries

2.1 Page Sharing

Sharing memory between processes can serve two different aims. It can be used as an inter-process communication mechanisms between two co-operating processes and it can be used for reducing memory footprint by avoiding replicated copies of identical contents. This paper focuses on the latter use.

When using *content-aware sharing*, identical pages are identified by the disk location the contents of the page is loaded from. This is the traditional form of sharing in an operating system, which is used for sharing the text segment of executable files between processes executing it and when using shared libraries [26]. Context-aware sharing has been suggested in early operating systems, such as Multics [42] and TENEX [13], and is implemented in all current major operating systems. This approach has also been suggested within the context of virtualisation hypervisors, such as Disco [15] and Satori [39].

Content-based page sharing, also called *memory de-duplication*, is a more aggressive form of page sharing. When using de-duplication, the system scans the active memory, identifying and coalescing unrelated pages with identical contents. De-duplication is implemented in the VMware ESX [54, 55] and PowerVM [17] hypervisors, and has also been implemented in Linux [6] and in Windows [33].

As memory pages can be shared between non co-operating processes, the system must protect the contents of the pages to prevent malicious processes from modifying the shared contents. To achieve this, the system maps shared pages as copy-on-write [13, 40]. Read operations on copy-on-write pages are permitted whereas write operations cause a CPU trap. The system software, which gains control of the CPU during the trap, copies the contents of the shared page, maps the copied page into the address space of the writing process and resumes the process.

While copy-on-write protects shared pages from modifications, it is not fully transparent. The delay introduced when modifying a shared page can be detected by processes, leading to a potential information leak attack. Such attacks have been implemented within virtualised environments for creating covert channels [58], for OS fingerprinting [44] and for detection of applications and data in other guests [49].

2.2 Cache Architecture

In addition to sharing memory pages, processes running on the same processor share the processor caches. Processor caches bridge the gap between the processing speed of modern processors and the data retrieval speed of the memory. Caches are small banks of fast memory in which the processor stores values of recently accessed memory cells. Due to locality of reference, recently used values tend to be used again. Retrieving these values from the cache saves time and reduces the pressure on the main memory.

Modern processors employ a cache hierarchy consisting of multiple caches. For example, the cache hierarchy of the Core i5-3470 processor, shown in Fig. 1, consists of three cache levels: L1, L2 and L3.

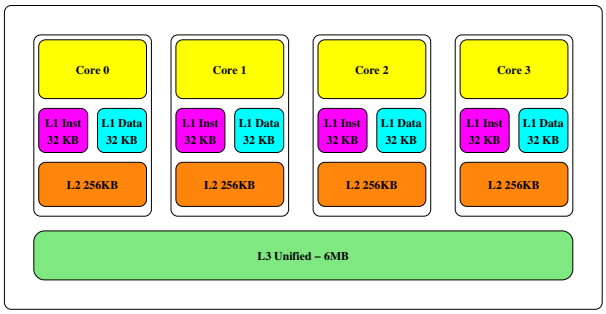


Figure 1: Intel Ivy Bridge Cache Architecture

The Core i5-3470 processor has four processing units called *cores*. Each core has a 64KB L1 cache, divided into a 32KB data cache and a 32KB instruction cache. Each core also has a 256KB L2 cache. The four cores

share a 6MB L3 cache, also known as the Last-Level Cache, or LLC.

The unit of memory in a cache is a *line* which contains a fixed number of bytes. A cache consists of multiple *cache sets* each of which stores a fixed number of cache lines. The number of cache lines in a set is the cache *associativity*. Each memory line can be cached in any of the cache lines of a single cache set. The size of cache lines in the Core i5-3470 processor is 64 bytes. The L1 and L2 caches are 8-way associative and the L3 cache is 12-way associative.

An important feature of the LLC in modern Intel processors is that it is an inclusive cache. That is, the LLC contains copies of all of the data stored in the lower cache levels. Consequently, flushing or evicting data from the LLC also remove said data from all other cache levels of the processor. Our attack exploits this cache behaviour.

Retrieving data from memory or from cache levels closer to memory takes longer than retrieving it from cache levels closer to the core. This difference in timing has been exploited for side-channel attacks. Side-channel attacks target information that an implementation of an algorithm leaks through its interaction with its environment. To exploit the timing difference, an attacker sets the cache to a known state prior to a victim operation. It can, then, use one of two methods to deduce information on the victim's operation [43]. The first method is measuring the time it takes for the victim to execute the operation. As this time depends on the state of the cache when the victim starts the operation, the attacker can deduce the cache sets accessed by the victim and, therefore, learn information on the victim [5, 9, 57]. The second approach is for the attacker to measure the time it takes for the attacker to access data after the victim's operation. This time is dependent on the cache state prior to the victim operation as well as on the changes the victim operation caused in the cache state [1, 2, 4, 14, 19, 47, 61].

Most prior work on cache side-channel attacks relies on the victim and spy executing within the same processing core. One reason for that is that many of the attacks suggested require the victim to be stopped while the spy performs the attack. To that aim, the attack is combined with an attack on the scheduler that allows the spy process to interrupt and block the victim.

Another reason for attacking within the same core is that the attacks focus on the L1 cache level, which is not shared between cores. The large size of the LLC hinders attacks both because setting it to a known state takes longer than with smaller caches and because the virtual memory used by the operating system masks the mapping of memory addresses to cache sets. Furthermore, as most of the memory activity occurs at the L1 cache level, less information can be extracted from LLC activity. Some prior works do use the LLC as an information leak channel [46, 47, 60]. However, due to the cache size, these channels have a low bandwidth.

We now proceed to describe the RSA encryption.

2.3 RSA

RSA [48] is a public-key cryptographic system that supports encryption and signing. Generating an encryption system requires the following steps:

- Randomly selecting two prime numbers p and q and calculating $n = pq$.
- Choosing a public exponent e. GnuPG uses $e = 65537$.
- Calculating a private exponent $d \equiv e^{-1} \pmod{(p-1)(q-1)}$.

The generated encryption system consists of:

- The public key is the pair (n, e).
- The private key is the triple (p, q, d).
- The encrypting function is $E(m) = m^e \bmod n$.
- The decrypting function is $D(c) = c^d \bmod n$.

CRT-RSA is a common optimisation for the implementation of the decryption function. It splits the secret key d into two parts $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, computes two parts of the message: $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$. m is then computed from m_p and m_q using Garner's formula [25]:

$$h = (m_p - m_q)(q^{-1} \bmod p) \bmod p$$
$$m = m_q + hq$$

To compute the encryption and decryption functions, GnuPG versions before 4.1.14 and the related `libgcrypt` before version 1.5.3 use the square-and-multiply exponentiation algorithm [28]. Square-and-multiply computes $x = b^e \bmod m$ by scanning the bits of the binary representation of the exponent e. Given a binary representation of e as $2^{n-1}e_{n-1} + \cdots 2^0 e_0$, square-and-multiply calculates a sequence of intermediate values $x_{n-1}, \ldots, x_0$ such that $x_i = b^{\lfloor e/2^i \rfloor} \bmod m$ using the formula $x_{i-1} = {x_i}^2 b^{e_{i-1}}$. Figure 2 shows a pseudo-code implementation of square-and-multiply.

As can be seen from the implementation, computing the exponent consists of sequence of Square and Multiply operations, each followed by a Modulo Reduce. This sequence corresponds directly with the bits of the exponent. Each occurrence of Square-Reduce-Multiply-Reduce within the sequence corresponds to a bit whose value is 1. Occurrences of Square-Reduce that are not

```
 1 function exponent(b, e, m)
 2 begin
 3   x ← 1
 4   for i ← |e| − 1 downto 0 do
 5     x ← x^2
 6     x ← x mod m
 7     if (e_i = 1) then
 8       x ← xb
 9       x ← x mod m
10     endif
11   done
12   return x
13 end
```

Figure 2: Exponentiation by Square-and-Multiply

followed by a Multiply correspond to bits whose values are 0. Consequently, a spy process that can trace the execution of the square-and-multiply exponentiation algorithm can recover the exponent.

As GnuPG uses the CRT-RSA optimisation, the spy process can only hope to extract d_p and d_q. However, for an arbitrary message m, $(m - m^{ed_p})$ is a multiple of p. Hence, knowing d_p (and, symmetrically, d_q) is sufficient for factoring n and breaking the encryption [16].

3 The FLUSH+RELOAD Technique

The FLUSH+RELOAD technique is a variant of PRIME+PROBE [51] that relies on sharing pages between the spy and the victim processes. With shared pages, the spy can ensure that a specific memory line is evicted from the whole cache hierarchy. The spy uses this to monitor access to the memory line. The attack is a variation of the technique suggested by Gullasch et al. [29], which include adaptations for use in multi-core and in virtualised environments.

A round of attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The spy, then, waits to allow the victim time to access the memory line before the third phase. In the third phase, the spy reloads the memory line, measuring the time to load it. If during the wait phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take significantly longer. Figure 3 (A) and (B) show the timing of the attack phases without and with victim access.

As shown in Fig. 3 (C), the victim access can overlap the reload phase of the spy. In such a case, the victim access will not trigger a cache fill. Instead, the victim will use the cached data from the reload phase. Consequently, the spy will miss the access.

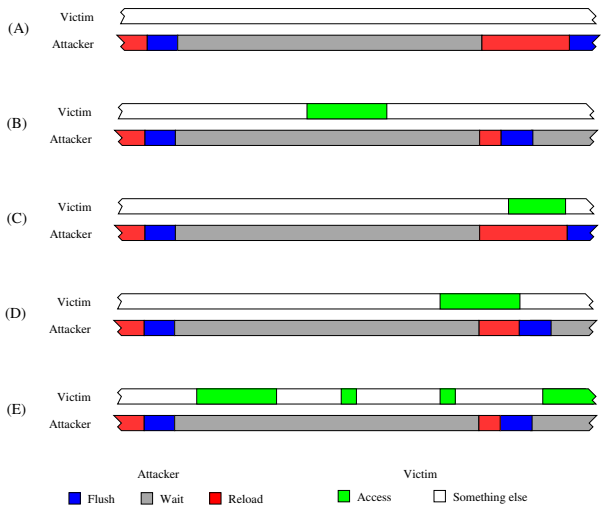


Figure 3: Timing of FLUSH+RELOAD. (A) No Victim Access (B) With Victim Access (C) Victim Access Overlap (D) Partial Overlap (E) Multiple Victim Accesses

A similar scenario is when the reload operation partially overlaps the victim access. In this case, depicted in Fig. 3 (D), the reload phase starts while the victim is waiting for the data. The reload benefits from the victim access and terminates faster than if the data has to be loaded from memory. However, the timing may still be longer than a load from the cache.

As the victim access is independent of the execution of the spy process code, increasing the wait period reduces the probability of missing the access due to an overlap. On the other hand, increasing the wait period reduces the granularity of the attack.

One way to improve the resolution of the attack without increasing the error rate is to target memory accesses that occur frequently, such as a loop body. The attack will not be able to discern between separate accesses, but, as Fig. 3 (E) shows, the likelihood of missing the loop is small.

Several processor optimisations may result in false positives due to speculative memory accesses issued by the victim's processor [34]. These optimisations include data prefetching to exploit spatial locality and speculative execution [52]. When analysing the attack results, the attacker must be aware of these optimisations and develop strategies to filter them.

Our implementation of the attack is in Figure 4. The code measures the time to read the data at a memory address and then evicts the memory line from the cache. This measurement is implemented by the inline assembly code within the `asm` command.

The assembly code takes one input, the address, which is stored in register `%ecx`. (Line 16.) It returns the time to read this address in the register `%eax` which is stored

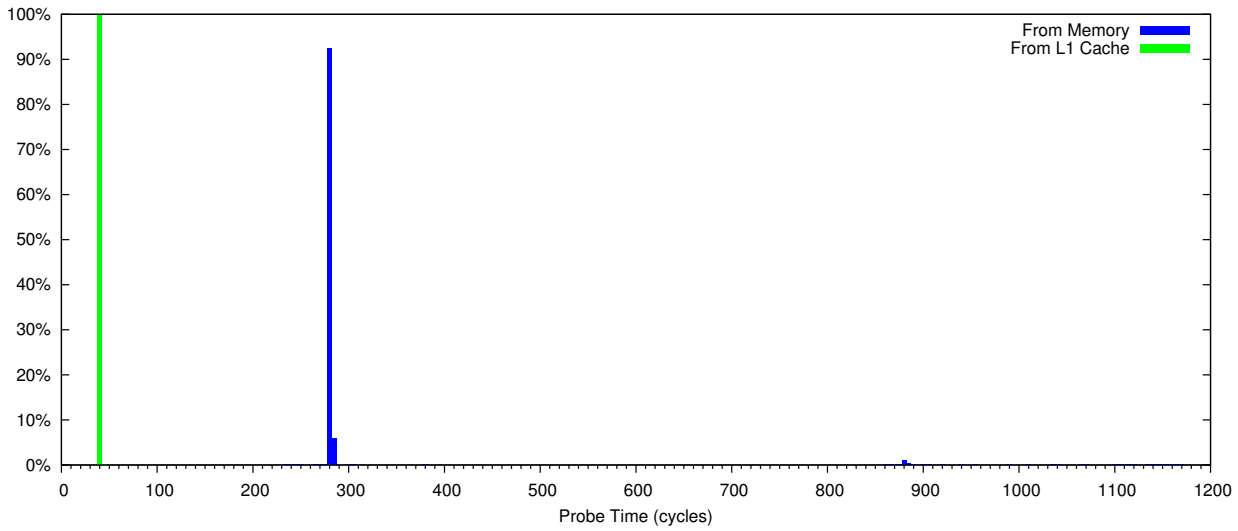


Figure 5: Distribution of Load Times.

```
 1  int probe(char *adrs) {
 2    volatile unsigned long time;
 3
 4    asm __volatile__ (
 5      "  mfence             \n"
 6      "  lfence             \n"
 7      "  rdtsc              \n"
 8      "  lfence             \n"
 9      "  movl %%eax, %%esi  \n"
10      "  movl (%1), %%eax   \n"
11      "  lfence             \n"
12      "  rdtsc              \n"
13      "  subl %%esi, %%eax  \n"
14      "  clflush 0(%1)      \n"
15      : "=a" (time)
16      : "c" (adrs)
17      :  "%esi", "%edx");
18    return time < threshold;
19  }
```

Figure 4: Code for the FLUSH+RELOAD Technique

in the variable `time`. (Line 15.)

Line 10 reads 4 bytes from the memory address in `%ecx`, i.e. the address pointed by `adrs`. To measure the time it takes to perform this read, we use the processor's time stamp counter.

The `rdtsc` instruction in line 7 reads the 64-bit counter, returning the low 32 bits of the counter in `%eax` and the high 32 bits in `%edx`. As the times we measure are short, we treat it as a 32 bit counter, ignoring the 32 most significant bits in `%edx`. Line 9 copies the counter to `%esi`.

After reading the memory, the time stamp counter is read again. (Line 12.) Line 13 subtracts the value of the counter before the memory read from the value after the read, leaving the result in the output register `%eax`.

The crux of the technique is the ability to evict specific *memory lines* from the cache. This is the function of the `clflush` instruction in line 14. The `clflush` instruction evicts the specific memory line from *all* the cache hierarchy, including the L1 and L2 caches of all cores. Evicting the line from all cores ensures that the next time the victim accesses the memory line it will be loaded into L3.

The purpose of the `mfence` and `lfence` instructions in lines 5, 6, 8 and 11 is to serialise the instruction stream. The processor may execute instructions in parallel or out of order. Without serialisation, instructions surrounding the measured code segment may be executed within that segment.

The `lfence` instruction performs partial serialisation. It ensures that load instructions preceding it have completed before it is executed and that no instruction following it executes before the `lfence` instruction. The `mfence` instruction orders all memory access, fence instructions and the `clflush` instruction. It is not, however, ordered with respect to other instructions and is, therefore, not sufficient to ensure ordering.

Intel recommends using the serialising instruction `cpuid` for that purpose [45]. However, in virtualised environments the hypervisor emulates the `cpuid` instruction. This software emulation takes too long (over 1,000 cycles) to provide the fine granularity required for the attack.

Line 18 compares the time difference between the two `rdtsc` instructions against a predetermined threshold. Loads shorter than the threshold are presumed to be served from the cache, indicating that another process has accessed the memory line since it was last flushed

from the cache. Loads longer than the threshold are presumed to be served from the memory, indicating no access to the memory line.

The threshold used in the attack is system dependent. To find the threshold for our test systems, we used the measurement code of the probe in Listing 4 to measure load times from memory and from the L1 cache level. (To measure the L1 times we removed the `clflush` instruction in line 14.) The results of 100,000 measurements of each on an HP Elite 8300 with an i5-3470 processor, running CentOS 6.5 are presented in Figure 5.

Virtually all loads from the L1 cache measure 44 cycles. (Note that this measure includes an overhead for the `rdtsc` and the fence instructions and is, therefore, much longer than a single load instruction.) Loads from memory show less constant timing. Over 98% of those take between 270 and 290 cycles. The rest are mostly spread around 880 cycles with about 200 loads measured 1140–1175 cycles. No loads from memory measured less than 200 cycles.

The timings of load operations depend on both the system architecture and the software environment. For example, on a Dell PowerEdge T420 with Xeon E5-2430 processors, loads from L1 take between 33 and 43 cycles and loads from memory take around 230 cycles. On the same architecture, within a KVM [37] guest, about 0.02% of the loads from memory take over 6,000 cycles. We believe these are caused by hypervisor activity.

The L1 measurements underestimate the probe time for data that the victim accesses. In an attack, data the victim accesses is read from the L3 cache. Intel documentation [34] states that the difference is between 22 and 39 cycles. Based on the measurement results and the Intel documentation we set the threshold to 120 cycles.

To use the FLUSH+RELOAD technique the spy and the victim processes need to share both the cache hierarchy and memory pages. In a non-virtualised environment, to share the cache hierarchy, the attacker needs the ability to execute software on the victim machine. The attacker, however, does not need elevated privileges on the victim machine. For a virtualised environment, the attacker needs access to a guest co-located on the same host as the victim guest. Techniques for achieving co-location are described by Ristenpart et al. [47]. Identifying the OS and software version in co-resident guests has been dealt with in past research [44, 49].

For sharing memory pages in system that use content-aware sharing, the attacker needs read access to the attacked executable or shared libraries. In systems that support de-duplication the attacker needs access to a copy of the attacked files. De-duplication will coalesce pages from these copies with pages from the attacked files.

4 Attacking GnuPG

In this section we describe how we use the FLUSH+RELOAD technique to extract the components of the private key from the GnuPG implementation of RSA.

We tested the attack on two hardware platforms: an HP Elite 8300, which features an Intel Core i5-3470 processor and 8GB DDR3-1600 memory and a Dell PowerEdge T420, with two Xeon E5-2430 processors and 32GB DDR3-1333 memory. On each hardware platform we experimented with two scenarios. The same-OS scenario tests the attack between two unrelated processes in the same operating system while the cross-VM scenario demonstrates that the attack works across the virtual machine isolation boundary in virtualised environments.

The same-OS tests use CentOS 6.5 Linux running on the hardware. The spy and the victim execute as two processes within that system. To achieve sharing, the spy `mmaps` the victim's executable file into the spy's virtual address space. As the Linux loader maps executable files into the process when executing them, the spy and the victim share the memory image of the mapped file. On the Dell machine we set the CPU affinity of the processes to ensure that both the victim and the spy execute on the same physical processor. We do let the processes float between the cores of the processor.

For the cross-VM scenario we used two different hypervisors: VMware ESXi 5.1 on the HP machine and Centos 6.5 with KVM on the Dell machine. In each hypervisor we created two virtual machines, one for the victim and the other for the spy. The virtual machines run CentOS 6.5 Linux. In this scenario, the spy `mmaps` a copy of the victim's executable file. Sharing is achieved through the page de-duplication mechanisms of the hypervisors. As in the same-OS scenario, on the Dell machine we set the CPU affinity of the virtual machines to ensure execution on the same physical processor.

When a pages is shared, all of the page entries in the virtual address spaces of the sharing processes map to the same physical page. As the LLC is physically tagged, entries in the cache depend only on the physical address of the shared page with no dependency on the virtual addresses in which the page is mapped. Consequently, we do not need to take care of the virtual to physical address mapping and the attack is oblivious to some diversification techniques, such as Address Space Layout Randomization (ASLR) [50].

The approach we take is to trace the execution of the victim program. For that, the spy program applies the FLUSH+RELOAD technique to memory locations within the victim's code segment. This, effectively, places probes within the victim program that are triggered whenever the victim executes the code in the probed memory lines. Tracing the execution allows the

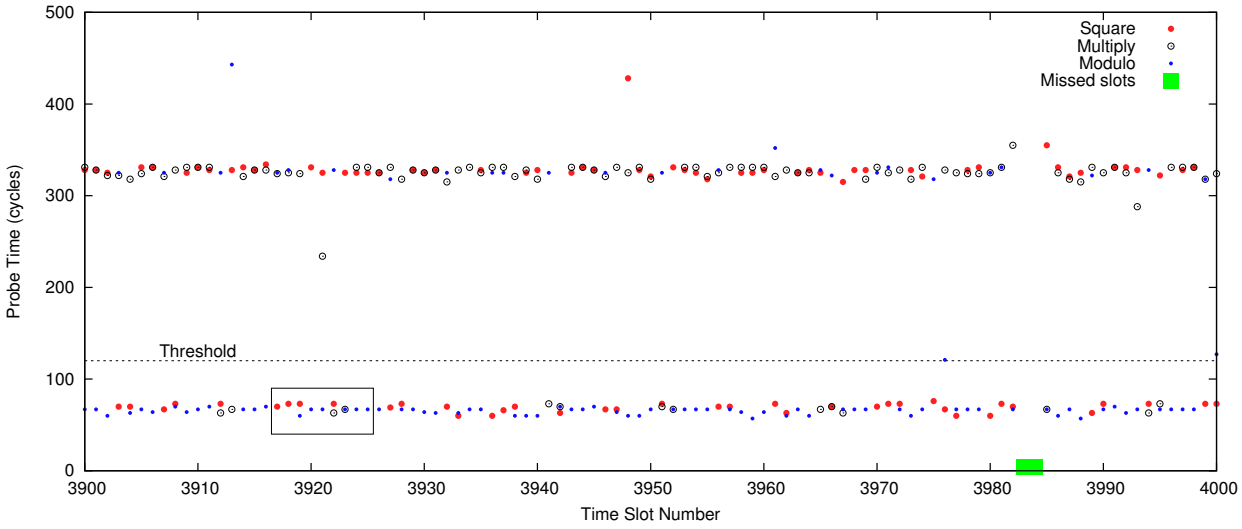


Figure 6: Time measurements of probes

spy program to infer the internal state of the victim program.

To implement the trace, the spy program divides time into fixed slots of 2,500 cycles each. In each slot it probes one memory line of the code of each of the square, multiply and modulo reduce calculations. To increase the chance of a probe capturing the access, we selected memory lines that are executed frequently during the calculation. Furthermore, to reduce the effect of speculative execution, we avoided memory lines near the beginning of the respective functions. After probing the memory lines, the spy program flushes the lines from the cache and busy waits to the end of the time slot.

We used the default build of the gpg program, which includes optimisation at -O2 level and which leaves the debugging symbols in the executable. We use the debugging symbols to facilitate the mapping of source code lines to memory addresses. In most distributions, the GnuPG executable is stripped and does not include these symbols. Attacks against stripped executables would require some reverse engineering [20] to recover this mapping. As the debugging symbols are not loaded in run time, these do not affect the victim's performance.

Measurement times for 100 time slots of the GnuPG signing with a 2,048 bit key are displayed in Figure 6. In each time slot, the spy flushes and then measures the time to read the memory lines in the Square, Multiply and Reduce functions. Measurements under the threshold indicate victim access to the respective memory lines. The exponentiations for signing takes a total of 15,690 slots or about 18ms. The CRT components used for exponentiation are 1,022 and 1,023 bits long.

Figure 7 is an enlarged view of the boxed section

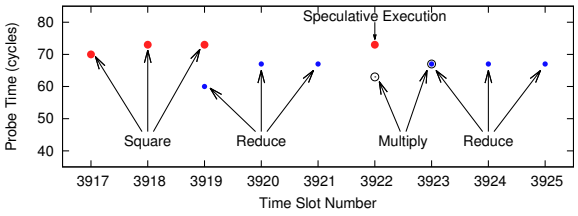


Figure 7: Section of Fig. 6

in Fig. 6. As the displayed area is below the threshold, the diagram only displays the memory lines that were retrieved from the cache, showing the activity of the GnuPG encryption. The steps of the exponentiation are clearly visible in the diagram. For example, between time slots 3,917 and 3,919 the victim was calculating a square, Time slots 3,919–3,921 are for modulo reduce calculation, multiplication in slots 3,922–3,923, and another modulo reduce in 3,923–3,925. A sequence of Square-Reduce-Multiply-Reduce indicates that during these time slots the victim was processing a set bit.

Figure 7 also demonstrates the effects of speculative execution. To improve performance, the processors tries to predict future behaviour of the program. When predicting the behaviour of the test of the bit value (Line 7 in Fig. 2), the processor does not know the value of the bit. Instead of waiting for the value to be calculated, the processor speculates that the bit might be clear and starts bringing memory lines required for the square calculation into the cache. As a result, cache lines that are part of the square calculation in Line 5 are brought into the cache, and are captured by the spy.

We have witnessed speculative execution on both the

HP and the Dell machines. Moving the probes to cache lines closer to the end of the probed functions eliminates the effects of speculative execution on the HP machine. However, speculative execution is still evident on the Dell machine.

By recognising sequences of operations, an attacker can recover the bits of the exponent. Sequences of Square-Reduce-Multiply-Reduce indicate a set bit. Sequences of Square-Reduce which are not followed by Multiply indicate a clear bit. For example, in Fig. 6, between time slots 3,903 and 3,906 the calculated sequence is Square-Reduce, which is followed by a Square, indicating that in these time slots the victim was processing a clear bit.

Continuing throughout Fig. 6 we find that the bit sequence processed in this sample is 0110011010011. Table 1 shows the time slots corresponding to each bit.

Table 1: Time Slots for Bit Sequence

Seq.	Time Slots	Value
1	3,903–3,906	0
2	3,907–3,916	1
3	3,917–3,926	1
4	3,927–3,931	0
5	3,932–3,935	0
6	3,936–3,945	1
7	3,946–3,955	1

Seq.	Time Slots	Value
8	3,956–3,960	0
9	3,961–3,969	1
10	3,970–3,974	0
11	3,975–3,979	0
12	3,980–3,988	1
13	3,989–3,998	1

System activity may cause the spy to miss time slots. The spy identifies missed time slot by noting jumps in the cycle counter. For example, In the run used for generating Fig. 6, the spy missed time slots 3,983 and 3,984. In this instance, the missed bits were not enough to hide the information on the bit processed during these time slots. However, if more slots are missed, data on bits of the private key exponent will be lost resulting in capture errors.

To measure the prevalence of capture errors, we used our spy program to observe and capture 1,000 signatures on each of the test configurations. We used a single invocation of a spy program to capture all the signatures in each system configuration. The GnuPG victim was executed from a shell in another window. Except for ensuring that the spy executes while running the signatures, the executions of the spy and of GnuPG are not synchronised.

For each observed signature, the spy outputs a text line representing the observed probes in each time slot. We used a shell script to parse this output and compared the results against the ground truth. The results are summarised in Table 2 and in Fig. 8. (For clarity, we trim Fig. 8 at 30% and 100 erroneous bits. A total of 15 samples have capture errors of more than 100 bits and the probability of no errors for the HP-CentOS configuration is 33%.)

Table 2: Statistics on Bit Errors in Capture

Hardware	HP Elite 8300		Dell PowerEdge T420	
Software	CentOS	VMware	CentOS	KVM
Average	1.41	26.55	25.12	66.12
Median	1	25	24	65
Max	15	196	96	190

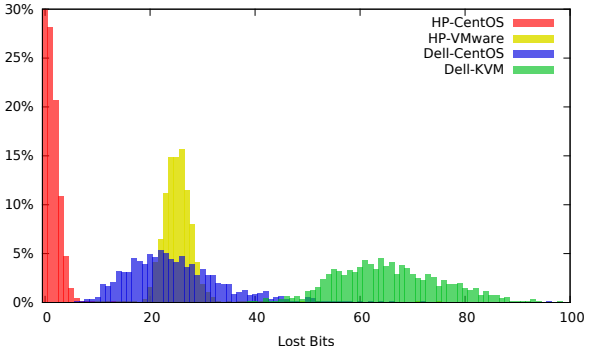


Figure 8: Distribution of Bit Errors in Capture

The shell script overestimates the number of errors. For example, due to the missing time slots, the script does not identify bit 12 in Table 1. We have manually inspected a few samples of capture output and estimate that manual inspection can reduce the number of errors by 25%-50%. Yet, the use of an automated script allows us to examine a large number of results.

On the HP machine we observe better results and significantly less noise than on the Dell machine. We believe this to be a consequence of the more advanced optimisations of the Xeon processor of the Dell machine. On each machine, results for the same-OS configuration are better than those for the cross-VM attack due to the added processing of the virtualisation layer.

Even accounting for the better results expected from manual inspection, the number of errors may be too big for a naïve brute force attack. Several strategies can be used to reduce the search space and to recover the private key. One such strategy is to rely on the nature of CRT-RSA exponentiation. As discussed in Section 2.3, an attacker only needs to recover one of the CRT components to break the encryption. By attacking the CRT component that has less errors, the attacker can reduce the search space to a more manageable size. Table 3 and Fig. 9 show the distribution of erroneous bits in the better captured CRT component in each signature. As these demonstrate, the search space is significantly reduced.

Several algorithms have been suggested for recovering the RSA exponent from partial information on the exponent bits [30, 31, 46]. These algorithms require between 27% and 70% of the bits of the exponent to recover the system key. While our attack reveals over 90% of the bits, it does not always recover the positions of

Table 3: Statistics on Bit Errors in the Better Captured CRT Component

Hardware	HP Elite 8300		Dell PowerEdge T420	
Software	CentOS	VMware	CentOS	KVM
Average	0.20	11.75	7.11	28.66
Median	0	12	6	28
Max	4	68	26	47

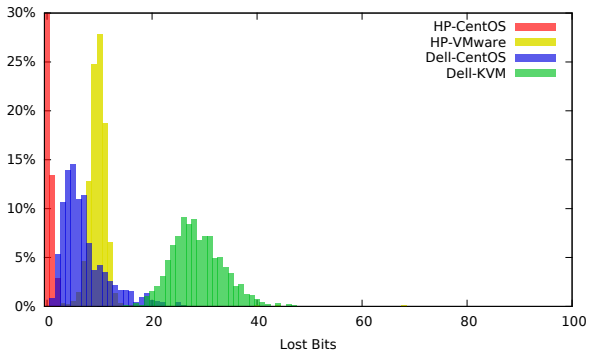


Figure 9: Distribution of Bit Errors in the Better Captured CRT Component

these bits. E.g. when a sequence of about 10 time slots is missed, this sequence can cover either one set bit or two clear bits. The attacker cannot, therefore, determine the bit positions of the following bits. Further research is required to determine whether these algorithms can be adapted to the data our attack recovers.

Another approach for recovering the key is to combine data from multiple signatures. As the positions of errors in each capture are independent, there is a small likelihood that any two captures will have errors in the same bit positions. To test this approach we manually merged the output of several pairs of observations of the spy under the Dell cross-VM scenario. When merging random pairs, we had at most a one bit error in the merged results. When merging the worst capture for the Dell cross-VM scenario with a random capture, the merged results had six bit errors, all of them in one of the CRT components and all have been identified during the process as potential errors. We, therefore, conclude that by observing two signatures, the attacker can recover the private key.

While the attack is very effective in recovering exponent bits, it does have some limitations. For the attack to work, the spy and the victim must execute on the same physical processor. For our testing, we set the processor affinity on the multi-processor system. However, in a real attack scenario the attack depends on the system scheduler.

When performing the tests, the spy and the victim were the only load on the system. Such a scenario is not representative of a real system where multiple processes are running. We expect such load to create noise that will affect the quality of capture. Furthermore, for a load that includes multiple parallel instances of GnuPG, the spy will be unable to distinguish between memory access of each instance and will be unable to recover any data.

Another limitation is the length of the secret key. On the Dell machine, probing three memory locations takes about 2,200 cycles. Hence, the attack cannot work with time slots shorter than that. With shorter key lengths, time slots of 2,200 cycles or more do not provide enough resolution to trace the victim. Consequently, recovering the private key is more difficult with shorter keys, supporting the results of Walter [56].

5 Mitigation Techniques

The attack presented here is a real, immediate threat to computer security. It, therefore, raises the very pertinent question of countermeasures. The FLUSH+RELOAD attack relies on a combination of four factors for its operation: data flow from sensitive data to memory access patterns, memory sharing between the spy and the victim, accurate, high-resolution time measurements and the unfettered use of the `clflush` instruction. Preventing any of these blocks the attack.

The lack of permission checks for using the `clflush` instruction is a weakness of the X86 architecture. Consequently, the most complete solution to the problem is to limit the power of the `clflush` instruction. The main use of the `clflush` instruction is to enforce memory coherence, e.g. when using devices that do not support memory coherence [34]. Another potential use of the instruction is to control the use of the cache for improving program performance, e.g. by flushing lines that the program knows it will not require. However, we are not aware of any actual use of the instruction for this purpose.

As the first use is, clearly, a system function and the second is based on the assumption that no other process has access to the data, we suggest restricting the use of `clflush` to memory pages to which the process has write access and to memory pages to which the system allows `clflush` access. This access control could be implemented by adding memory types that restrict flush access to the PAT (Page Attribute Table) [35, chap. 11].

The ARM architecture [7] also includes instructions to evict cache lines. However, these instructions can only be used when the processor is in an elevated privilege mode. As such, the ARM architecture does not allow user process to selectively evict memory lines and the FLUSH+RELOAD is not applicable in this architecture.

Our attack seems not to work on contemporary AMD processors, such as the A10-6800K and Opteron 6348. The code in Fig. 5 returns the same result with and without the `clflush` instruction. Replacing the second

`rdtsc` instruction (Line 12) with the similar `rdtscp` instruction fixes this issue, however, two problems prevent the use of the technique. The first problem is that data seems to linger in the cache for some time after being evicted. The second problem is that the attack does not capture accesses from other processes. A possible explanation for this behaviour is that the AMD caches are non-inclusive, i.e. data in L1 does not need to also be in L2 or L3, as is the case with the Intel caches. Consequently, evicting data from the LLC does not, necessarily, evicts it from the L1 caches of other cores. Processes executing on other cores can access data in the L1 cache without triggering a load from memory to the LLC. The attack does work on older AMD processors, such as the Opteron 2212.

Hardware based countermeasures, such as those described above cannot provide an immediate solution to the problem. They will take time to develop and will not protect existing hardware. Consequently, for immediate mitigation of the attack, software-based solutions are required.

Another possible solution is preventing sharing between the spy and the victim. Preventing page sharing between processes provides protection against the FLUSH+RELOAD attack. However, this approach goes against the trend of increased sharing in operating systems and virtualisation hypervisors. Completely eliminating page sharing would significantly increase the memory requirements of modern operating systems and is, therefore, unlikely to be a feasible solution. As a partial solution, it may be possible to avoid sharing of sensitive code by changing the program loader. Another partial solution is disabling page de-duplication, which prevents using the FLUSH+RELOAD attack between co-hosted guests in a virtualised system. This approach is recommended for public compute clouds which offer the implied promise that guests cannot interfere with each other.

Software diversification [24] is a collection of techniques that permute the locations of objects within the address spaces of processes. While most of these techniques were originally developed as a protection against memory corruption attacks, some of them can be used to prevent sharing and, consequently, to mitigate the FLUSH+RELOAD attack. More specifically, in virtualised environments, static reordering of code and data [12,24,36] can be used to create unique copies of programs in each virtual machines. As these copies are not available outside the specific virtual machine, pages of the program are not de-duplicated and sharing is prevented. Diversifying the program at run time [22] can prevent sharing of the program text even when the attacker has access to the binary file. As discussed above, the FLUSH+RELOAD technique is oblivious to the virtual to physical address mapping. Consequently, diversification techniques that rely on permuting the virtual address mapping of code pages, such as [50, 59], do not provide any protection against the attack.

FLUSH+RELOAD, like other side-channel attacks, relies on the availability of a high-resolution clock. Reducing the resolution of the clock or introducing noise to clock measurement [32, 53] can be an used as a countermeasure against the attack. The main limitation of this approach is that the attacker can use other methods for generating high resolution clocks. Examples include using data from the network or running a 'clock' process in a separate execution core.

Irrespective of the measures described above, cryptographic software should be protected against the attack. Following our disclosure [18, 38], the GnuPG team released GnuPG version 1.4.14 and `libgcrypt` version 1.5.3. These mitigate the attack using the square-and-multiply-always [21] algorithm, shown in Listing 10. The algorithm executes the square and the multiply steps for each bit, but ignores the result of the multiply step for bits of value 0.

```
function exponent(b, e, m)
begin
  x ← 1
  for i ← |e| − 1 downto 0 do
    x ← x^2
    x ← x mod m
    x' ← xb
    x' ← x' mod m
    if (e_i = 1) then
      x = x'
    endif
  done
  return x
end
```

Figure 10: Exponentiation by Square-and-Multiply-Always

When introducing instructions with no effect, care should be taken to prevent the compiler from optimising these away. In the case of the GnuPG fix, the optimiser cannot know that the added addition does not have side-effects. With the possibility of side-effects, the optimiser takes a conservative approach and invokes the function.

The implementation still contains a small section of code that depends on the value of the bit, which could, theoretically, be exploited by a cache side-channel attack. However, due to speculative execution, the processor is likely to access the section irrespective of the value of the bit. Furthermore, as this section is short and is smaller than a cache line, it is likely to fit within the same cache line as the preceding or following code. Hence, we believe that this implementation protects against the FLUSH+RELOAD attack.

This fix, however, does not protect against other forms of side-channel attack. In particular, the code is likely to be vulnerable to Branch Prediction Analysis [3]. Furthermore, as access patterns to data depend on the values of the exponent bits, the code is likely to be vulnerable to PRIME+PROBE attacks [51,61]. Like FLUSH+RELOAD, these side-channel attacks rely on data flow from secret exponent bits to memory access patterns. These attacks can be prevented by using *constant time* exponentiation, where the sequence of instructions and memory locations accessed are fixed and do not depend on the value of the exponent bits. Techniques for constant time computation have been explored in the NaCl cryptographic library [10]. The pattern of accesses to memory lines of the OpenSSL [41] implementation of RSA exponentiation is not dependent on secret exponent bits. Consequently, even though the implementation is not constant time [11], it is not vulnerable to our attack.

Constant time computation is not, however, a panacea for the problem of side-channel attacks. FLUSH+RELOAD can be applied no extract secret data from non cryptographic software. For such software, the performance costs of constant-time computation are unreasonable, hence other solutions are required.

6 Related Work

Several works have pointed out that page sharing exposes guests to information leakage, which can be exploited for implementing covert channels [58], OS fingerprinting [44] and for detecting applications and data in other guests [49]. These works exploit the copy-on-write feature of page sharing. Copy-on-write introduces a significant delay when a page is copied. Hence, by timing write operations on pages, a spy can deduce the existence of pages with identical contents in other guests. As page de-duplication is a slow process, all these attacks have a very low resolution.

Using a cache side-channel to trace the execution of a program is not a new idea [1, 2, 4, 14, 19, 29, 61]. In all of these attacks, the victim and the spy must share the execution core, either by using hyper-threading or by interleaving the execution of the victim and the spy on the same core.

Gullasch et al. [29] describes an attack on AES which traces the victim's access to the S-Boxes. Our work builds on the attack technique presented by Gullasch et al. and extends it in two ways. Gullasch et al. only applies the attack on a time-shared core and does not exploit the eviction from a shared LLC. Our attack exposes the use of a shared LLC and demonstrates that the technique can be used across cores. Additionally, Gullasch et al. uses the `cpuid` instruction to synchronise the instruction stream whereas we use fence instructions. In virtualised environments, the `cpuid` is emulated in software and this emulation takes over 1,000 cycles. With two `cpuid` instructions in each probe, the Gullasch et al. probe spans over 2,500 cycles. As our attack requires three probes within 2,500 cycles, the resolution of the Gullasch et al. code is is not high enough for implementing our cross-VM attack.

The attack in Zhang et al. [61] specifically targets virtualised environments, extracting the private ElGamal [23] key of a GnuPG decryption executing in another guest. The attack depends on a weakness in the scheduler of the Xen hypervisor [8]. The granularity of the attack is one probe in 50,000 cycles, limiting the minimum size of victim key that can be captured. The modulus in the paper is 4,096 bits long. The attack has low signal to noise ratio, and requires the use of filtering. Even with this filtering and the large modulus, the attack requires six hours of constant decryption to recover the key.

Weiß et al. [57] also describes cache timing attack in a virtualised environment. The attack is an adaptation of Bernstein's attack [9] that relies on the short constant communication time between domains in the L4 kernel.

7 Conclusions

In this paper we describe the FLUSH+RELOAD technique and how we use it to extract GnuPG private keys across multiple processor cores and across virtual machine boundaries.

It is hard to overstate the severity of the attack, both in virtualised and in non-virtualised environments. GnuPG is a very popular cryptographic package. It is used as the cryptography module of many open-source projects and is used, for example, for email, file and communication encryption. Hence, vulnerable versions of GnuPG are not safe for multi-tenant systems or for any system that may run untrusted code.

While significant, the attack on GnuPG is only a demonstration of the power of the FLUSH+RELOAD technique. The technique is generic and can be used to monitor other software. It can be used to devise other types of attacks on cryptographic software. It can also be used against other types of software. For example, it could be used to collect statistical data on network traffic by monitoring network handling code or it could monitor keyboard drivers to collect keystroke timing information.

Hence, while the GnuPG team has fixed the vulnerability in their software, their fix does not address the broader threat exposed by this paper.

The FLUSH+RELOAD technique exploits the lack of restrictions on the use of the `clflush` instruction. Not restricting the use of the instruction is a security weakness of the Intel implementation of the X86 architecture. This enables processes to interact using read-only pages.

Addressing this weakness requires a hardware fix, which, unless implemented as a microcode update, will not be applicable to existing hardware.

Preventing page sharing also blocks the FLUSH+RELOAD technique. Given the strength of the attack, we believe that the memory saved by sharing pages in a virtualised environment does not justify the breach in the isolation between guests. We, therefore, recommend that memory de-duplication be switched off.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Thomas Ristenpart, for their valuable comments and support.

This research was performed under contract to the Defence Science and Technology Organisation (DSTO) Maritime Division, Australia.

References

[1] ACIIÇMEZ, O. Yet another microarchitectural attack: exploiting I-Cache. In *Proceedings of the ACM Workshop on Computer Security Architecture* (Fairfax, Virginia, United States, November 2007), P. Ning and V. Atluri, Eds., pp. 11–18.

[2] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems* (Santa Barbara, California, United States, April 2010), S. Mangard and F.-X. Standaert, Eds., pp. 110–124.

[3] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the Second ACM Symposium on Information, Computer and Communication Security* (Singapore, March 2007), pp. 312–320.

[4] ACIIÇMEZ, O., AND SCHINDLER, W. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the Cryptographers' Track at the RSA Conference* (San Francisco, California, United States, April 2008), T. Malkin, Ed., pp. 256–273.

[5] ACIIÇMEZ, O., SCHINDLER, W., AND KOÇ, Ç. K. Cache based remote timing attacks on the AES. In *Proceedings of the Cryptographers' Track at the RSA Conference* (San Francisco, California, United States, February 2007), M. Abe, Ed., pp. 271–286.

[6] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium* (Montreal, Quebec, Canada, July 2009), pp. 19–28.

[7] *ARM Architecture Reference Manual*, ARMv7-A and ARMv7-R ed., 2012.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art or virtualization. In *Proceedings of the Ninteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, New York, United States, October 2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 164–177.

[9] BERNSTEIN, D. J. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, April 2005.

[10] BERNSTEIN, D. J., LANGE, T., AND SCHWABE, P. The security impact of a new cryptographic library. In *Proceedings of the Second International Conference on Cryptology and Information Security in Latin America* (Santiago, Chile, October 2012), A. Hevia and G. Neven, Eds., pp. 159–176.

[11] BERNSTEIN, D. J., AND SCHWABE, P. A word of warning. CHES 2013 Rump Session, August 2013.

[12] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium* (Washington, DC, United States, August 2003), pp. 105–120.

[13] BOBROW, D. G., BURCHFIEL, J. D., MURPHY, D. L., AND TOMLINSON, R. S. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM 5*, 3 (March 1972), 135–143.

[14] BRUMLEY, B. B., AND HAKALA, R. M. Cache-timing template attacks. In *Advances in Cryptology - ASIACRYPT 2009* (2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 667–684.

[15] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 4 (November 1997), 412–447.

[16] CAMPAGNA, M., AND SETHI, A. Key recovery method for CRT implementation of RSA. Report 2004/147, IACR Cryptology ePrint Archive, 2004.

[17] CERON, R., FOLCO, R., LEITAO, B., AND TSUBAMOTO, H. *Power Systems Memory Deduplication*. IBM, September 2012.

[18] CERT vulnerability note vu#976534: L3 cpu shared cache architecture is susceptible to a Flush+Reload side-channel attack. `http://www.kb.cert.org/vuls/id/976534`, October 2013.

[19] CHEN, C., WANG, T., KOU, Y., CHEN, X., AND LI, X. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *The Journal of Systems and Software 86*, 1 (2013), 100–107.

[20] CIPRESSO, T., AND STAMP, M. Software reverse engineering. In *Handbook of Information and Communication Security*, P. Stavroulakis and M. Stamp, Eds. Springer, 2010, ch. 31, pp. 659–696.

[21] CORON, J.-S. Resistence against differential power analysis for elliptic curve cryptosystems. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems* (Worcester, Massachusetts, United States, August 1999), Ç. K. Koç and C. Paar, Eds., pp. 292–302.

[22] CURTSINGER, C., AND BERGER, E. D. STABILIZER: Statistically sound performance evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, United States, March 2013), pp. 219–228.

[23] ELGAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31*, 4 (July 1985), 469–472.

[24] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. H. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems* (Cape Code, Massachusetts, United States, May 1997), pp. 67–72.

[25] GARNER, H. L. The residue number system. *IRE Transactions on Electronic Computers EC-8*, 2 (June 1959), 140–147.

[26] GINGELL, R. A., LEE, M., DANG, X. T., AND WEEKS, M. S. Shared libraries in SunOS. In *USENIX Conference Proceedings* (Phoenix, Arizona, United States, Summer 1987), pp. 131–145.

[27] GNU Privacy Guard. http://www.gnupg.org, 2013.

[28] GORDON, D. M. A survey of fast exponentiation methods. *Journal of Algorithms 27*, 1 (April 1998), 129–146.

[29] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games — bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, United States, may 2011), pp. 490–595.

[30] HENINGER, N., AND SHACHAM, H. Reconstructing RSA private keys from random key bits. In *Proceedings of the 29th Annual International Cryptology Conference (CRYPTO 2009)* (Santa Barbara, California, United States, August 2009), S. Halevi, Ed., pp. 1–17.

[31] HERMANN, M., AND MAY, A. Solving linear equations modulo divisors: On factoring given any bits. In *Advances in Cryptology - ASIACRYPT 2008* (Melbourne, Australia, December 2008), vol. 5350 of *Lecture Notes in Computer Science*, pp. 406–424.

[32] HU, W.-M. Reducing timing channels with fuzzy time. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, United States, May 1991), pp. 8–20.

[33] HUFFMAN, C. Memory combining in Windows 8 and Windows Server 2012. http://blogs.technet.com/b/clinth/archive/2012/11/29/memory-combining-in-windows-8-and-windows-server-2012.aspx, November 2012.

[34] INTEL CORPORATION. *Intel 64 and IA-32 Architecture Optimization Reference Manual*, April 2012.

[35] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, March 2013.

[36] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the Annual Computer Security Applications Conference* (Miami Beach, Florida, United States, December 2006), pp. 339–348.

[37] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. **kvm**: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, June 2007), vol. one, pp. 225–230.

[38] KOCH, W. GnuPG 1.4.14 released. http://lists.gnupg.org/pipermail/gnupg-announce/2013q3/000330.html, July 2013.

[39] MIŁOŚ, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference* (San Diego, California, United States, June 2009).

[40] MURPHY, D. L. Storage organization and mamagement in TENEX. In *Proceedings of the Fall Joint Computer Conference, AFIPS'72, Part I* (Anaheim, California, United States, December 1972), pp. 23–32.

[41] OPENSSL. http://www.openssl.org.

[42] ORGANICK, E. I. *The Multics System: An Examination of Its Structure*. The MIT Press, 1972.

[43] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, November 2005.

[44] OWENS, R., AND WANG, W. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *Proceedings of the 30th IEEE International Performance Computing and Communicatons Conference* (Orlando, Florida, United States, November 2011), S. Zhong, D. Dou, and Y. Wang, Eds., IEEE, pp. 1–8.

[45] PAOLONI, G. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, September 2010.

[46] PERCIVAL, C. Cache missing for fun and profit. http://www.daemonology.net/papers/htt.pdf, 2005.

[47] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communication Security* (Chicago, Illinois, United States, November 2009), E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., pp. 199–212.

[48] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (February 1978), 120–126.

[49] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory deduplication as a threat to the guest. In *Proceedings of the 2011 European Workshop on System Security* (Salzburg, Austria, 2011).

[50] The PaX project. http://pax.grsecurity.net/.

[51] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks in AES, and countermeasures. *Journal of Cryptology 23*, 2 (January 2010), 37–71.

[52] UHT, A. K., AND SINDAGI, V. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture* (Ann Arbor, Michigan, United States, November 1995), pp. 313–325.

[53] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating fine grained timers in Xen. In *Proceedings of the ACM Workshop on Cloud Computing Security* (Chicago, Illinois, United States, October 2011), C. Cachin and T. Ristenpart, Eds., pp. 41–46.

[54] VMWARE INC. *Understanding Memory Resource Management in VMware ESX Server*. Palo Alto, California, United States, 2009.

[55] WALDSPURGER, C. A. Memory resource management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, Massachusetts, United States, December 2002), D. E. Culler and P. Druschel, Eds., pp. 181–194.

[56] WALTER, C. D. Longer keys may facilitate side channel attacks. In *Selected Areas in Cryptography* (2004), M. Matsui and R. J. Zuccherato, Eds., vol. 3006 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 42–57.

[57] WEISS, M., HEINZ, B., AND STUMPF, F. A cache timing attack on AES in virtualization environments. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security* (Bonaire, February 2012), A. D. Keromytis, Ed.

[58] XIAO, J., XU, Z., HUANG, H., AND WANG, H. A covert channel construction in virtualized environments. In *Proceedings of the 19th ACM Conference on Computer and Communication Security* (Raleigh, North Carolina, United States, October 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., pp. 1040–1042.

[59] XU, J., KALBARCZYK, Z., AND IYER, R. K. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems* (Florence, Italy, October 2003), pp. 260–269.

[60] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the ACM Workshop on Cloud Computing Security* (Chicago, Illinois, United States, October 2011), C. Cachin and T. Ristenpart, Eds., pp. 29–40.

[61] ZHANG, Y., JULES, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communication Security* (Raleigh, North Carolina, United States, October 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., pp. 305–316.

Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks

Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk
Horst Görtz Institute for IT-Security, Ruhr-University Bochum
<{christopher.meyer, juraj.somorovsky, eugen.weiss, joerg.schwenk}@rub.de>

Sebastian Schinzel <schinzel@fh-muenster.de>
Department of Computer Science, Münster University of Applied Sciences

Erik Tews <erik@datenzone.de>
European Center for Security and Privacy by Design,Technische Universität Darmstadt

Abstract

As a countermeasure against the famous Bleichenbacher attack on RSA based ciphersuites, all TLS RFCs starting from RFC 2246 (TLS 1.0) propose "to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks".

In this paper we show that this objective has not been achieved yet (cf. Table 1): We present four new Bleichenbacher side channels, and three successful Bleichenbacher attacks against the *Java Secure Socket Extension (JSSE)* SSL/TLS implementation and against hardware security appliances using the *Cavium NITROX SSL accelerator chip*. Three of these side channels are timing-based, and two of them provide the first timing-based Bleichenbacher attacks on SSL/TLS described in the literature. Our measurements confirmed that all these side channels are observable over a switched network, with timing differences between 1 and 23 microseconds. We were able to successfully recover the `PreMasterSecret` using three of the four side channels in a realistic measurement setup.

TLS impl.	Side channel	Queries & Efficiency	
		Queries	Time
OpenSSL	timing	$O(2^{40})$	n.a.
JSSE	error message	177,000	12 h
JSSE	timing	18,600	19.5 h
Cavium	timing	7371	41 h

Table 1: Overview on Bleichenbacher side channels and attacks. In case of timing based side channels, *Queries* denotes the number of queries sent to the Bleichenbacher oracle $\mathcal{O}$ (see below); the actual number of requests sent to the TLS server (and thus the attack duration) depend on the network quality. Even though we found timing differences in the OpenSSL implementation, the attack revealed not to be practical due to the weakness of the oracle.

1 Introduction

SSL/TLS is, due to its enormous importance, a major target for attacks. During the last years, novel attack techniques (targeting the TLS Record Layer) have been discovered (see e.g. [21]). However, one of the most famous attacks is still Bleichenbacher's chosen-ciphertext attack on the TLS handshake [5], exploiting side channels of the RSA decryption process (see Section 3). Formal models don't cover this attack: The first full security proof of the TLS-RSA handshake [17] assumes that the RSA decryption implementation is ideal without any side channels.

Bleichenbacher's Attack. Bleichenbacher's attack is an adaptive chosen-ciphertext attack on the RSA PKCS#1 v1.5 encryption padding scheme (denoted by PKCS#1 in the following). The only prerequisite for the attack is the presence of a side channel at the TLS server which allows to distinguish PKCS#1 compliant from non-compliant ciphertexts. An attacker with access to such a side channel can proceed as follows: He records the TLS handshake of the target connection, and extracts the RSA-PKCS#1 encrypted `ClientKeyExchange` message c. Then he iteratively creates new ciphertexts $c', c'', \ldots$ from c. These are sent to the TLS server as part of a new handshake, and the server's responses are observed. With each successful query, i.e. a query c^* which is PKCS#1 compliant, the attacker can reduce the interval in which the original plaintext is located in. He repeats these steps until the interval only contains one integer, thus decrypting the ciphertext c. Daniel Bleichenbacher successfully applied this attack to SSL 3.0 [5] in 1998.

In three of the four presented attacks we are dealing with timing based side channels, so we have to repeat measurements to statistically eliminate random noise. In the following, we use an abstraction to deal with this fact: A Bleichenbacher *oracle* $\mathcal{O}$ receives a candidate ciphertext c^* as input and makes use of a side channel (e.g. by

repeating measurements) to finally output whether c^* is PKCS#1 compliant or not (see Figure 3).

Countermeasures. Soon after the publication of the original Bleichenbacher attack in 1998, error messages were unified and the TLS standards introduced the following countermeasure: If the decrypted message structure is not compliant, the TLS server generates a random `PreMasterSecret`, and performs all subsequent handshake computations with this value.[1] This countermeasure was described in TLS versions 1.0 [9] and 1.1 [10]. TLS 1.2 [11] improves this by prescribing that a random number must *always* be generated, independently of the PKCS#1 compliance of the incoming ciphertext. This should ensure equal processing times for compliant and non-compliant ciphertexts.

Novel Side Channels. In this paper we analyze several widely used TLS implementations for their vulnerability against Bleichenbacher attacks and show that the implemented countermeasures are not sufficient: We describe four new Bleichenbacher oracles, and analyze their sources (see Table 1). Additionally, the strength of these oracles is evaluated and three of these oracles are shown to be strong enough to mount Bleichenbacher attacks in practice. This finally led to the decryption of previously recorded SSL/TLS sessions.

The first side channel is caused by an implementation bug in the *Java Secure Socket Extension (JSSE)* – Java's built-in SSL/TLS implementation. In JSSE a different error message can be triggered if the two most significant bytes are PKCS#1 compliant, but the `PreMasterSecret` shows up to be of invalid length. We were able to successfully exploit this and decrypt a `PreMasterSecret` with a few thousand queries.

The second side channel is based on conspicuous timing differences in the *OpenSSL* implementation during PKCS#1 processing. The source of this side channel is hard to determine: Our working assumption suggests that it is based on the additional time consumption of choosing a random value. Following the description of Bleichenbacher countermeasures in TLS versions 1.0 and 1.1, this random value is only generated if the decrypted `PreMasterSecret` is not PKCS#1 compliant. The timing difference (in the range of few microseconds) caused by the unequal treatment of random number generation (depending on the PKCS#1 compliance of the ciphertexts) may be the cause for this side channel. We were able to reliably measure a timing difference in the range of one microsecond over a LAN and to reliably detect plaintexts containing valid `PreMasterSecret` values.

The third side channel is based on the fact that Java's `Exception` handling and error processing can be a time consuming task: Whenever the resulting plaintext is not PKCS#1 compliant, an `Exception` is raised by *JSSE* forcing random `PreMasterSecret` generation. The resulting timing difference is significantly higher (in the range of 20 microseconds) and can be measured over a LAN. This qualifies the side channel for practical attacks under real-world conditions.

The fourth side channel was found in widely used *F5 BIG-IP* and *IBM Datapower* products which rely on the *Cavium NITROX SSL accelerator chip*. It allowed to distinguish invalid messages from messages starting with `0x??02` (where `0x??` represents an arbitrary byte). Since the original Bleichenbacher algorithm does not handle this case, we derived a novel variant of the algorithm and evaluated that it can decrypt 2048-bit ciphertexts with only 4700 queries to an oracle.

Contribution. The contributions of this paper can be summarized as follows:

- *Impact.* We analyze several widely used SSL/TLS implementations and identify four new Bleichenbacher side channels, three of them timing-based. We describe three successful Bleichenbacher attacks which completely break JSSE and NITROX based SSL/TLS accelerators.

- *Novelty.* We describe the first timing based Bleichenbacher attacks against a TLS implementation. We present a novel variant of the original Bleichenbacher algorithm to handle specific server behavior and show that this variant results in a much better attack performance.

- *Insight.* We show that `Exception` handling may cause large timing differences, measurable over a LAN. This observation is in general important for development of side channel free (cryptographic) implementations in object oriented languages.

- *Methodology.* Our research was conducted using a novel framework for SSL/TLS inspection and penetration, called T.I.M.E., which may be of independent interest.

Responsible Disclosure. All vulnerabilities were communicated to the vendors' security teams and sent together with fix proposals. They were fixed or are going to be fixed in the newest releases.

[1] This leads to a fatal error when checking the `ClientFinished` (because of different `PreMasterSecret` at client and server side), but it does not allow the attacker to distinguish valid from invalid ciphertexts based on server error messages.

2 SSL/TLS

The Secure Sockets Layer (SSL) protocol was invented 1994 by Netscape Communications, and later (1999) renamed to Transport Layer Security (TLS) by the IETF. It evolved to be the de facto standard for secure data transmission over the Internet and is mostly used, but not limited, to secure HTTP traffic.

SSL/TLS mainly consists of two components: the *Handshake Protocol* to negotiate security primitives and key material, and the *Record Layer* where the payload (HTTP, IMAP, ...) is encrypted and integrity protected.

Record Layer. The Record Layer is initiated with the `NULL` ciphersuite, where no cryptographic protection is applied at all. Then the handshake is executed, until the `ChangeCiperSpec` message is sent by one party. Immediately after sending this message, this party switches the Record Layer to the negotiated parameters (algorithms and keys) and enables the negotiated security algorithms.

Subsequently, all messages sent through the TLS channel are secured by the selected cipher suite algorithms and the computed key material. Regarding integrity and confidentiality the Record Layer relies on a MAC-then-PAD-then-Encrypt scheme ([22] gives a detailed overview on this topic and highlights the pitfalls). The payload data is integrity protected by a (keyed H)MAC, padded if required, and finally encrypted.

Handshake Protocol. This protocol is used to negotiate the cryptographic primitives and keys. The different primitives are bundled in *cipher suites*. A *cipher suite* defines the algorithms for *(a)* key exchange or key agreement, *(b)* encryption (and, if necessary, the mode of operation) and *(c)* MAC (Message Authentication Code). Thus, the cipher suite `TLS_RSA_WITH_DES_CBC_SHA` uses (a) RSA encryption for key exchange, (b) DES encryption in CBC mode for encryption and (c) a SHA-1 based HMAC for integrity to protect the payload.

Figure 1 illustrates a typical (RSA-based) handshake without mutual authentication, between a client and a server. All cipher suites supported by the client are listed in the `ClientHello` message, and one of these suites is chosen by the server in the `ServerHello` message. The server's public key for RSA encryption is sent in the `Certificate` message and the ciphertext of the `PreMasterSecret` chosen by the client is contained in the `ClientKeyExchange` message. After this message, both - client and server - are ready to switch to encrypted mode (by sending a `ChangeCiperSpec` message). The final two *`Client-/Server`*`Finished` messages (containing a cryptographic checksum over all previously exchanged handshake messages) are already encrypted.

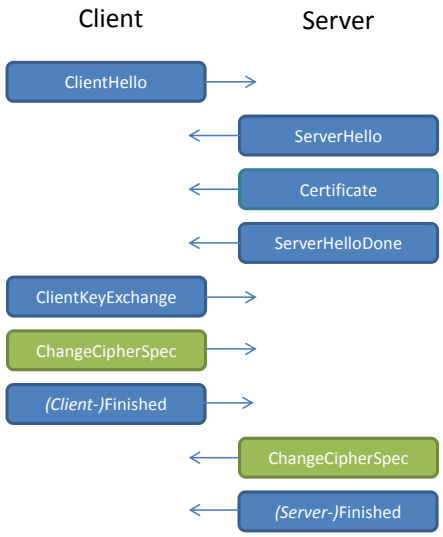


Figure 1: SSL/TLS handshake with RSA Key Exchange

Other Protocols. The *ChangeCipherSpec Protocol* is used to activate channel protection (switch to negotiated cipher suite and related key material), whereas the *Alert Protocol* is responsible for signalizing errors and failures.

Libraries and Appliances. The work presented in this paper focuses on the most common open source libraries and SSL/TLS appliances listed below.

OpenSSL.[2] As a widely used open source library OpenSSL is applied by many applications (such as the Apache Webserver's default module for SSL/TLS).

Java Secure Socket Extension (JSSE).[3] This library is the standard implementation of SSL/TLS for the Java platform, provided as part of the Java Runtime Environment. Java based applications are very likely to use it.

GnuTLS.[4] GnuTLS is another open source library for SSL/TLS available under GPL.

***IBM Datapower* and *F5 BIG-IP*.** These two products are widely used Web application firewalls and security appliances. Their SSL/TLS processing is handled using a *Cavium NITROX SSL accelerator chip*.

3 Bleichenbacher's Attack

In 1998, Daniel Bleichenbacher presented an adaptive chosen-ciphertext attack on protocols using the RSA PKCS #1 encryption standard [5]. He exemplarily applied his attack to the SSL v3.0 protocol. Through different error messages returned from the SSL server, Blei-

[2] `http://www.openssl.org`

[3] `http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html`

[4] `http://www.gnutls.org`

chenbacher was able to identify ciphertexts where the plaintext started with 0x00 02. Thus, he used the SSL server as a (partial) decryption oracle $\mathcal{O}$ and was able to decrypt an encrypted PreMasterSecret, from which all SSL/TLS session keys are derived [11]. Soon after this discovery, the error messages were unified in order to close this side channel. Later, the attack was reenabled by Klíma, Pokorný and Rosa [16] through a different side channel, fixed again and finally remained unexploitable for nearly 10 years.

In order to describe the basic attack, we will first give an overview of the PKCS#1 encryption padding scheme and its usage in SSL/TLS to secure the PreMasterSecret. Afterwards, the attack and the countermeasures are presented. Throughout this section we write $|a|$ to denote the byte-length of a string a, and $a||b$ to denote concatenation of a and b. We let (N,e) be an RSA public key, with corresponding secret key d.

3.1 PKCS#1 v1.5 Encryption Padding

PKCS#1 v1.5. The basic task of the PKCS#1 v1.5 encryption padding scheme is to prepend a random padding string PS ($|PS| > 8$) to a message k, and then apply the RSA encryption function:

1. The encrypter takes a message k and chooses a random, non-zero string PS, where $|PS| > 8$ and $|PS| = \ell - 3 - |k|$.
2. The cleartext block is $m = 00||02||PS||00||k$. By interpreting this string as an integer $m < N$,
3. the ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the decrypter first computes $m = c^d \bmod N$. Afterwards, it is checked whether the decrypted message m has a correct PKCS#1 format. This message $m = m_1||m_2||...||m_{|m|}$ is PKCS#1 compliant if ($x \geq 10$):

$$\begin{aligned} m_1 &= \texttt{0x00} \\ m_2 &= \texttt{0x02} \\ \texttt{0x00} &\notin \{m_3,\ldots,m_x\} \\ \texttt{0x00} &\in \{m_{x+1},\ldots,m_{|m|}\} \end{aligned} \tag{1}$$

PKCS#1 usage in TLS. In case of TLS, PKCS#1 is used for encapsulation of the PreMasterSecret exchanged during a handshake which consists of 48 bytes. The first two bytes of the PreMasterSecret contain a two-byte version number $maj||min$ (e.g., $maj = 0x03$, $min = 0x01$ for TLS 1.0). The remaining bytes are chosen by the client at random. Figure 2 gives an example of a PreMasterSecret (PMS) padded to be encrypted with a 2048-bit RSA key.

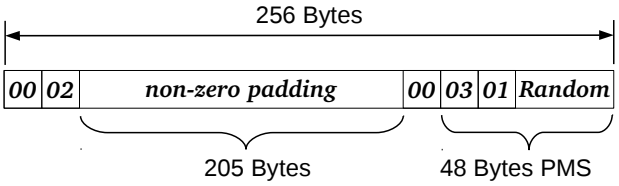


Figure 2: PKCS#1 padding applied to a PMS to be encrypted with a 2048-bit RSA key

We say that a PKCS#1 compliant message m is *TLS compliant* if:

$$\begin{aligned} |k| &= 48 \\ k_1||k_2 &= maj||min \end{aligned} \tag{2}$$

3.2 Basic Attack Idea.

Bleichenbacher's attack enables an adversary, who is in possession of a ciphertext c_0, to recover the encrypted plaintext m_0. The only prerequisite for this attack is the ability to access an oracle $\mathcal{O}$ that decrypts a ciphertext c and responds with 1 or 0, depending on whether the decrypted message m starts with 0x00 02 or not:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \mod N \text{ starts with } \texttt{0x00 02} \\ 0 & \text{otherwise.} \end{cases}$$

If the oracle answers with 1, the adversary knows that $2B \leq m \leq 3B-1$, where $B = 2^{8(\ell-2)}$. The algorithm is based on the malleability of the RSA encryption scheme which allows the following blinding:

$$c = (c_0 \cdot s^e) \bmod N = (m_0 s)^e \bmod N$$

The attacker queries the oracle with c. If the oracle responds with 0, the attacker increments s and repeats the previous step. Otherwise, the attacker learns that

$$2B \leq m_0 s - rN < 3B$$

for some r. This allows the attacker to reduce the set of possible solutions to

$$\frac{2B+rN}{s} \leq m_0 < \frac{3B+rN}{s}$$

By iteratively choosing new values for s, querying the oracle, and computing new r values, the attacker narrows down the interval which contains the original m_0 value. He repeats these steps until only one solution in the interval is left. We refer to the original paper [5] for details.

3.3 Impact of Oracle Type on Attack Performance

The oracle $\mathcal{O}$ needed for the attack can be based on different side channels. For example, it can be provided by a server responding with different error messages based on the PKCS#1 compliance. If the server identifies a message as PKCS#1 compliant, the attacker knows the message starts with 0x00 02.

Bleichenbacher tested his attack against an SSL server which strictly checked the PKCS#1 format (see Equation 1). He needed about one million messages to decrypt an arbitrary ciphertext (1024-bit RSA). However, the attack performance varies. Bleichenbacher's algorithm relies solely on the knowledge that the first two message bytes are equal to 0x00 02. If an oracle is constructed from an application which verifies only the first two bytes of the decrypted message (0x00 02), we get a very "strong" oracle and the attack performs well. On the other hand, if an application checks also different properties such as TLS protocol version conformity (see Equation 2), the oracle can respond with 0 even if the first two bytes are equal to 0x00 02 (e.g., if the extracted `PreMasterSecret` is of invalid length). Such a behavior leads to false negatives which slow down the attack performance. The oracle is "weak".

The oracle strength can be measured using a probability that the oracle responds with 1 when a given decrypted message starts with 0x00 02. Suppose $P(A)$ defines a probability that the first two bytes of the decrypted message are 0x00 02. $P(1|A)$ is a probability that the oracle answers with 1, in case that the decrypted message starts with 0x00 02. Suppose we work with a 1024 bit RSA key. For an oracle strictly checking the PKCS#1 compliance (first eight bytes do not contain 0x00, but one of the following 118 bytes contains 0x00), the probability can be computed as:

$$P^{1024}_{PKCS}(1|A) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{118}\right) \approx 0.36$$

Different oracle types and their impact on the attack performance were analyzed by Bardou et al. [4]. In addition, they improved Bleichenbacher's attack by a factor of four. An improved attack running with the discussed oracle needs about 15,000 queries to decrypt a PKCS#1 compliant message (more queries are needed to decrypt an arbitrary message).

3.4 Countermeasures

Due to its importance, Bleichenbacher's attack is directly addressed in the TLS standard [11]. The basic idea of the proposed countermeasure is to continue the processing with a randomly generated `PreMasterSecret` every time the message structure is invalid or decryption failed completely. This ensures unified error messages of the server. Algorithm 1 describes the implementation of this countermeasure as proposed in TLS 1.2 [11]:

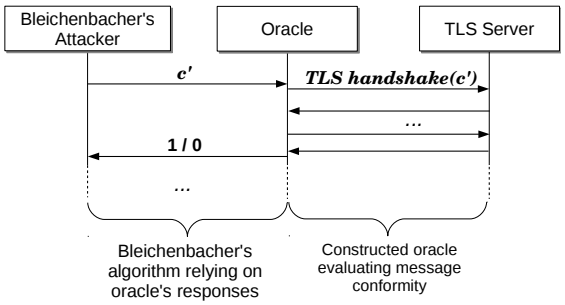


Figure 3: Bleichenbacher's attack algorithm relies on an oracle returning 1 or 0 according to the message validity.

Algorithm 1 A (simplified) countermeasure against Bleichenbacher's attack proposed in the TLS standard [11].

1: generate a random PMS_R
2: decrypt the ciphertext: $m := dec(c)$
3: **if** ($(m \neq 00||02||PS||00||k)$ OR $(|k| \neq 48)$
OR $(k_1||k_2 \neq maj||min)$ **then**
4: proceed with $PMS := PMS_R$
5: **else**
6: proceed with $PMS := k$
7: **end if**

This countermeasure ensures that *each* ciphertext decryption reveals a `PreMasterSecret` which is used in the handshake processing. Thus, the attacker cannot distinguish between valid and invalid ciphertexts. Note that a random `PreMasterSecret` is generated every time, independently from the ciphertext validity. This ensures equal processing times of valid and invalid ciphertexts.

4 SSL/TLS Penetration Testing

Given the importance of PKCS#1 format processing in SSL/TLS, it is important how Bleichenbacher countermeasures are implemented in real-world applications.

4.1 Attack Challenges

We investigate ways of turning a seemingly secure SSL/-TLS server into an oracle $\mathcal{O}$ suitable for Bleichenbacher's attack. The attack is sketched in Figure 3: The attacker communicates with $\mathcal{O}$ and suggests ciphertexts. $\mathcal{O}$ sends these ciphertexts to the server by performing a TLS handshake, evaluates its responses, and returns 1 or 0 according to the PKCS#1 conformity.

The oracle can be based on different side channels. First, *noisy* TLS servers responding with different error messages represent a direct oracle $\mathcal{O}_D$. Second, even if the server does not respond with different error messages, its processing logic can cause different timings while handling valid and invalid ciphertexts. These *silent* checks can be used to construct a timing oracle $\mathcal{O}_T$.

When constructing an oracle $\mathcal{O}$, we have to face the following challenges:

1. $\mathcal{O}$ must not respond with false positives: ciphertexts falsely identified as valid cause Bleichenbacher's algorithm to end up in a wrong internal state from which the algorithm cannot recover.

2. $\mathcal{O}$ should respond with as few false negatives as possible: valid ciphertexts falsely identified as invalid slow down the attack performance.

3. $\mathcal{O}$ should require as few requests as possible.

4.2 T.I.M.E.

This research was enabled by a new framework called T.I.M.E. - TLS Inspection Made Easy (for details see [20]). The framework implements a TLS client stack in Java with means to intercept the communication and TLS protocol flow at any time through predefined hookpoints. It allows altering TLS messages in an object based representation or, if necessary, even at bit level. This renders deep analysis of TLS, simulating complex attack scenarios, or trigger bugs only occurring in usually hard to provoke operation states possible. The framework proved to be well suited for the creation of a large amount of test cases, even in complex attack scenarios. The modularity allows a quick test case creation and automated testing for vulnerabilities of many different TLS implementations with comparably little effort. A comprehensive reporting engine eases the analysis even when working with large amounts of test cases and scanning targets.

Architecture. Figure 4 illustrates the T.I.M.E. architecture. It consists of the following main parts:

- SSL/TLS Stack and Network Stack handle the communication between the framework and the remote SSL/TLS server.

- The Attack Engine consists of different attack modules including one for Bleichenbacher's attack. It contains the attack logic and test cases for triggering different server behavior to identify bugs in the server's SSL/TLS stack.

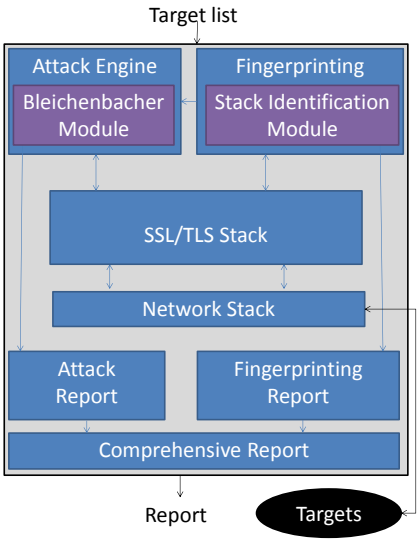


Figure 4: T.I.M.E. architecture

- The Fingerprinting Engine generates specifically formatted messages and triggers different server behavior which is analyzed to identify the SSL/TLS implementation and its version. The description of this engine is out of scope of this paper.

- The Reporting Module generates attack and fingerprinting reports.

The whole process of intercepting a running communication is event based. An application is able to register for events of interest, in this case e.g. the `ClientKeyExchange` message and `Alerts`. The workflow notifies each observer about occurring events. Once an observer is notified, the execution control is passed to this observer. The observer can manipulate the current message or internal states of the stack and return the control back to the workflow. The communication is paused until the observer returns control. Once returned the workflow continues immediately with processing.

The interaction between server, attack module and the handshake workflow of T.I.M.E. is illustrated in Figure 5.

The Bleichenbacher attack logic is built directly upon the stack and can be used to modify messages during the TLS handshake. The modified messages are used to trigger different server behavior. This allows to check for obvious vulnerabilities to Bleichenbacher's attack.

4.3 Test Environment

As we are performing timing attacks over a network, special care must be taken for the measurement setup. Measuring precise processing times from remote is challenging because of the jitter induced by busy network

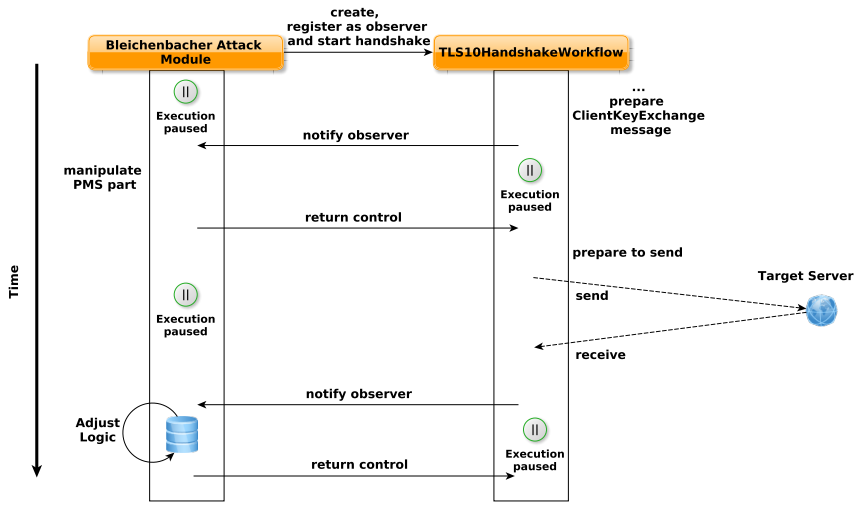


Figure 5: Interaction between the components. The Bleichenbacher Attack Module instantiates a `TLS10HandshakeWorkflow` object (part of the T.I.M.E. framework), registers as an observer for the `ClientKeyExchange` and `Alert` messages and finally starts the workflow. Every time one of these messages occurs the handshake is paused and the Bleichenbacher Attack Modules gains control. It either modifies the encrypted `PreMasterSecret` or analyzes the response message. Finally, it returns the control back to the workflow which continues with the handshake.

components, by the remote machine and by the measuring client. We also wanted to perform our attacks in a realistic scenario, in which the attacker has full control over the measuring machine, but only limited control over the network quality. We therefore ran the measurement machine with a stripped down Ubuntu 12.04 LTS Linux where we disabled CPU halting (boot parameter `idle=poll`) and CPU frequency scaling (fixing the CPU frequency using the `cpufreq` tools). Both settings are not uncommon in data centers that trade faster response times for higher power consumption. We used a Realtek 8139-based networking card with no support for interrupt coalescing. Note that this configuration likely optimizes the quality of the timing measurements, but it is not a necessary requirement. For a comprehensive analysis of hardware choices and configuration settings for timing measurements over networks see [8].

It is realistic to assume that the attacker has some limited control over the network. For example, if the connection from the attacker's machine to the target machine is of bad quality, the attacker can often rent (or compromise) a machine nearby the target machine and launch the attack from there (consider cloud-based scenarios). We therefore used a network setting in which the attacking and target machine are in the same (productive) University campus LAN connected through a Cisco Catalyst 2950 switch. This setting emulates the environment of a common co-location center or a cloud system where the attacker might even be able to rent a virtual machine that runs on the same hardware as the target machine [23].

If we use the attack module for triggering different TLS server messages, the whole T.I.M.E. tool set is placed on a single machine and communicates as a client with the remote TLS server. For timing measurement we had to act differently after we found out that T.I.M.E. provides no reliable base for highly fine grained time measurement. Thus, we decided to split the Bleichenbacher logic and the TLS logic into separate modules. Figure 6 illustrates this setup. On the left, we see the Bleichenbacher attack module that triggers and executes the attack. The Bleichenbacher logic generates new ciphertexts and hands it over to the measurement module.

To test if a TLS implementation has a suitable timing leak that allows the creation of a timing-based oracle, one has to measure the delay between the `ClientKeyExchange` message and the arrival of the `HANDSHAKE_FAILURE` message (the server performs PKCS#1 checking during this period). High precise timing measurement is not possible in Java (the JVM itself causes a significant *noise* which falsifies the results). Thus, we modified the lightweight *MatrixSSL* C implementation[5] to execute the TLS handshake and measure the timing delays in clock ticks by using the RDTSC assembler directive.

We used the timing analysis tool NetTimer[6] to evaluate the server response times. This tool implements a

[5] `http://www.matrixssl.org/`
[6] `http://sebastian-schinzel.de/nettimer`

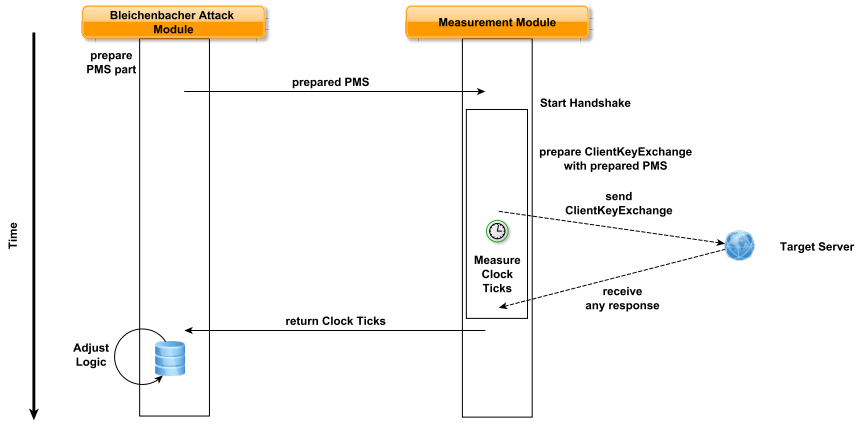


Figure 6: Architecture for measuring timing differences. The enhanced T.I.M.E. framework is split into two parts: The Bleichenbacher Attack Module and the Measurement Module based on the MatrixSSL library.

variant of Crosby's box hypothesis test, which was found to perform well for analyzing network delay measurements [8]. With this setup, we were able to reliably distinguish timing differences of a few hundred nanoseconds over a LAN with one thousand repeated measurements. This confirms that the findings of [8] not only hold for artificial UDP ping-pong protocols, but also for real-world TCP-based protocols.

4.4 Methodology

Our methodology during evaluation of Bleichenbacher's attack on a specific implementation can be summarized in the following steps.

Triggering Different Server Behaviors. In Section 3.1 we described how an encrypted ciphertext is processed on a TLS server. This process includes several validation and unpadding steps. If one of these steps is implemented incorrectly, a side channel might arise. Thus, we first implemented different T.I.M.E. test cases that aim to trigger different server behavior which could lead to a practical oracle $\mathcal{O}$. These test cases include:

1. A TLS compliant message, see Equation 2.
2. A PKCS#1 compliant message which is not TLS compliant, see Equation 1. Such a message can include a wrong TLS version number or a `PreMasterSecret` with an invalid length.
3. A non-PKCS#1 compliant message: Such a message can for example start with a non-zero byte or can be missing the `0x00` byte after the random padding of the message.

We cover all three cases and send the encrypted messages to the target TLS server and observe if the server responds with different error messages or timing behavior. As we analyze open source TLS frameworks, we are able to combine the automatic analysis of the T.I.M.E. framework with an additional source code review.

Analyzing Oracle Strength. We analyze if the discovered side channel can be used to construct a practical (*Strong*) Bleichenbacher oracle. This can be achieved by considering two factors. First, the probability that the oracle responds with 1 if the decrypted message starts with `0x00 02`. Second, in case of a timing oracle, how many server requests are needed to distinguish a valid from an invalid ciphertext.

Performing the Attack. In order to assess the practicability and performance of the attack using a constructed oracle, we use the oracle in a real attack execution and report on the number of oracle queries. For this purpose, we implemented the Bleichenbacher attack [5] as T.I.M.E. test case and extended it with the trimming and skipping holes methods from [4].

5 First Side Channel: Error Messages in JSSE

Automated evaluation of JSSE with T.I.M.E. revealed a new side channel which could be used to construct a noisy oracle $\mathcal{O}_{D-JSSE}$ leading to a successful Bleichenbacher attack. In general, the side channel is caused by an improper padding check and the subsequent `PreMasterSecret` processing. This behavior enabled us to force the server to respond with different alerts while processing differently formatted PKCS#1 messages: `INTERNAL_ERROR` and `HANDSHAKE_FAILURE`.

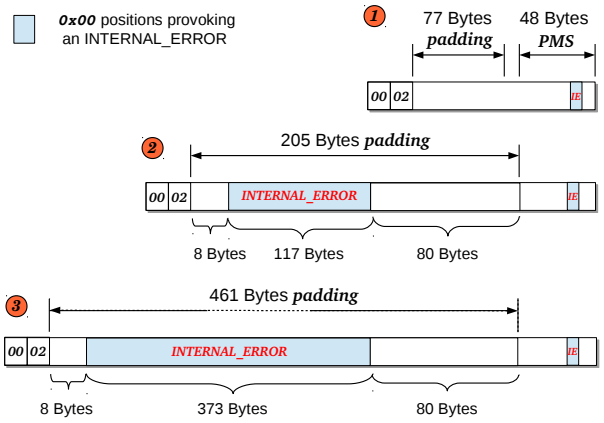


Figure 7: If a decrypted message contains a 0x00 byte preceded with non-0x00 bytes in at least one of the marked positions, JSSE responds with an INTERNAL_ERROR alert. The depicted messages are of 1024 (1), 2048 (2), and 4096 (3) bit length.

Side Channel Analysis. In the following, we analyze the attack on the server with a 2048 bit (256 bytes) key. Similar analysis could be applied to other key sizes.

Due to a fixed length of the PreMasterSecret (*PMS*), the padding string length can easily be determined to be 205 bytes (see Figure 2). These bytes must not include a 0x00 byte. The T.I.M.E. framework enabled us to test the JSSE implementation with specifically formatted messages. The analysis revealed that 0x00 bytes inserted at specific padding positions cause an internal ArrayIndexOutOfBoundsException leading to a different TLS alert message. The exception was caused when the PreMasterSecret length check was not correctly applied (cf. Algorithm 1, line 3). Propagation of the unchecked ArrayIndexOutOfBoundsException to the surface lead to the communication abort, the server responded with an INTERNAL_ERROR alert.

More precisely, our test revealed that changing either the first 8 or last 80 padding bytes led to a correct HANDSHAKE_FAILURE alert. Changing one of the remaining padding bytes to 0x00 caused a different INTERNAL_ERROR alert. This was caused by the MasterSecret computation initialized with a PreMasterSecret of an incorrect length. By applying 2048 bit RSA keys, the number of bytes causing an INTERNAL_ERROR alert is equal to 117 (depicted in Figure 7). In case of 4096 bit keys, this number is equal to 373 (see Figure 7).

In addition to the positions described above in 2048 and 4096 bit long ciphertexts, our analysis revealed that there is also a chance to attack 1024 bit ciphertexts directly. Independently of the applied key size, the server responded with an INTERNAL_ERROR if the second to last byte ($m_{|m|-1}$) contained 0x00 and the preceding bytes do not contain 0x00.

The different alert messages offered a new oracle $\mathcal{O}_{D-JSSE}$ responding with 1 (INTERNAL_ERROR) or 0 (HANDSHAKE_FAILURE) according to the structure of the decrypted PreMasterSecret.

Oracle Strength. In the following, we evaluate the probability for 2048 and 4096 bit random messages starting with 0x00 02 to contain a structure causing an INTERNAL_ERROR alert. Let n be the byte size of the PKCS#1 message and $|PMS|$ the PreMasterSecret length. The number of bytes provoking an INTERNAL_ERROR can be derived as:

$$x = n - 3 - |PMS| - 8 - 80.$$

Let us consider that the first two message bytes are 0x00 02. The probability that the following 8 padding bytes are non-zero and at least one of the following x bytes becomes 0x00 (and thus the server responds with an INTERNAL_ERROR alert) is:

$$P_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^x\right)$$

For key sizes of 2048 and 4096 bits (256 and 512 bytes) it results in:

$$P^{2048}_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^{117}\right) \approx 0.356$$

$$P^{4096}_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^{373}\right) \approx 0.744$$

This means that a JSSE server ($\mathcal{O}_{D-JSSE}$) using a 2048 bit RSA key responds with a probability of $P^{2048}_{D-JSSE}(1|A) \approx 35.6\%$ with 1 (INTERNAL_ERROR), if the decrypted PreMasterSecret message starts with 0x00 02. In case of using 4096 bit keys, the oracle is even more permissive. It responds with a probability of $P^{4096}_{D-JSSE}(1|A) \approx 74.4\%$ if the message starts with 0x00 02. These probabilities suggest a low number of false negatives, leading to an efficient Bleichenbacher attack.

On the other hand, when applying 1024 bit long RSA keys, $\mathcal{O}_{D-JSSE}$ is much less permissive. It responds with an INTERNAL_ERROR only if 0x00 is positioned just before the last byte. Thus, the probability $P^{1024}_{D-JSSE}(1|A)$ can be computed as:

$$P^{1024}_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^{124} \cdot \left(\frac{1}{256}\right) \approx 0.0024$$

	Mean	Median
2048 bit RSA key	176,797	37,399
4096 bit RSA key	73,710	27,744

Table 2: Number of required queries to execute an optimized Bleichenbacher's attack on a JSSE server using 2048 bit and 4096 bit RSA keys.

Attack Evaluation. We used this oracle to perform a Bleichenbacher attack – the experiment was repeated 1,000 times. Results of this evaluation confirm the findings of our theoretical analysis from the previous section: Executing the attack using a less restrictive oracle with a 4096 bit RSA key leads to fewer oracle queries. We needed about 177,000 queries to a JSSE server applying 2048 bit keys and about 74,000 queries to a JSSE server applying 4096 bit keys. See Table 2 for details.

We performed full `PreMasterSecret` recovery attacks against a TLS server working with 2048 bit keys. With our T.I.M.E. framework we were able to send about 3.85 server queries per second. Thus, sending 177,000 requests lasted about 12 hours. The attack was performed on localhost.[7]

Performance evaluation of an oracle using 1024 bit keys resulted in hundreds of millions of oracle queries. This is caused by the high restrictiveness of $\mathcal{O}_{D-JSSE}$ when applying keys of this length.

Mitigation. We communicated this problem to the Oracle Security response team and the bug was assigned CVE-2012-5081. The attack is fixed with the *Oracle Java SE Critical Patch October 2012 – Java SE Development Kit 6, Update 37 (JDK 6u37)*.

6 Second Side Channel: Timing Differences in OpenSSL

The discovery of the aforementioned vulnerability in JSSE motivated to investigate the source code of open source SSL/TLS frameworks. We reviewed JSSE, GnuTLS and OpenSSL and found that they do not implement the countermeasure against Bleichenbacher's attack as proposed by the TLS 1.2 specification [11].

Side Channel Analysis. The countermeasure against this attack is mostly implemented as depicted in Algorithm 1. The important observation is that the random key is generated if, and only if, the received key is not

[7]Improving the T.I.M.E. sending performance would result in much faster attack executions. This was however not the primary goal of our work.

Algorithm 2 Improper implementation of the countermeasure against Bleichenbacher's attack (suggested by TLS 1.0 and TLS 1.1) possibly causing a timing side channel in all the analyzed implementations.

1: decrypt the ciphertext: $m := dec(c)$
2: **if** ($(m \neq 00||02||PS||00||k)$ OR $(|k| \neq 48)$
OR $(k_1||k_2 \neq maj||min)$ **then**
3: generate a random PMS_R
4: proceed with $PMS := PMS_R$
5: **else**
6: proceed with $PMS := k$
7: **end if**

TLS compliant (see Equation 2). Thus, the random key generation and the assignment create a new timing side channel that leaks information about the TLS compliance of a received `PreMasterSecret`. These processing steps have independently been observed and criticized by Matthew Green [18].

Oracle Strength. *Timing Reliance.* We tested the timing differences between valid and invalid ciphertexts with OpenSSL 0.98. Figure 8 shows the filtered results of our timing analysis over a LAN with 5,000 measurements. The results suggest that we can distinguish TLS compliant and non-PKCS#1 compliant ciphertexts. We could achieve similar results for OpenSSL 1.01.

Even though the results clearly showed constant differences of about 1.5 microseconds, we are not sure if the root cause of these differences is additional random number generation. The OpenSSL code contained sev-

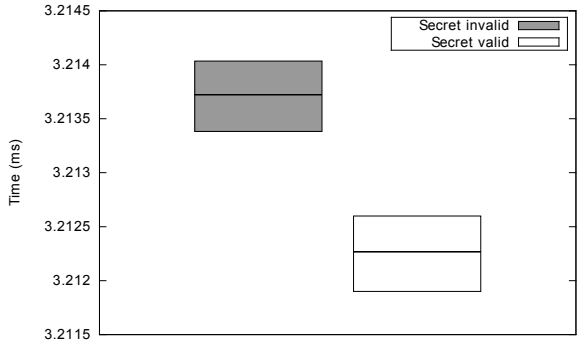


Figure 8: Timing measurement results for OpenSSL 0.98. The valid secret refers to a TLS compliant ciphertext. The invalid secret refers to a non-PKCS#1 compliant ciphertext. In the non-PKCS#1 compliant structure the first byte (which should be `0x00`) was altered to `0x08` to provoke a random number generation on the TLS server.

eral additional branches and loops in the PKCS#1 processing which could blur our results. The analysis of this problem showed up to be very difficult and related to compile flags. Despite this uncertainty, our measurements clearly show that a side channel exists.

Probability Analysis. The analyzed timing behavior can be used to construct an oracle

$$\mathcal{O}_{T-rand}(c) = \begin{cases} 1 & \text{TLS compliant} \\ 0 & \text{non-TLS compliant (with an additional random number generation)} \end{cases}$$

However, it does not lead to a practical attack. An oracle created from this timing leak is very "weak". It responds to an oracle request with 1 if, and only if, the decrypted ciphertext is TLS compliant (see Equation 2). For a 2048 bit key, the probability that an oracle responds with 1 in case that the decrypted message starts with `0x00 02` is very low:

$$P^{2048}_{T-rand}(1|A) = \left(\frac{255}{256}\right)^{205} \cdot \left(\frac{1}{256}\right)^{3} \approx 2.7 \cdot 10^{-8}$$

The reason is that 205 padding bytes must be non-zero and the following bytes must contain `0x00`$||maj||min$. See Figure 2.

Attack Evaluation. $\mathcal{O}_{T-rand}$ is very "weak" and did not allow to execute a practical Bleichenbacher attack. We were only able to estimate the number of oracle queries. According to Bleichenbacher and Bardou et al. [5, 4], the number of oracle queries for the complete attack can be computed as:

$$(2^{17} + 16 \cdot 256)/P_{T-rand} = 5 \cdot 10^{12}$$

Mitigation. The mitigation is described in RFC 5246 [11]. Algorithm 1 illustrates the correct processing: A random value should always be generated, *before* processing the decrypted data.

7 Third Side Channel: Internal Exception

We decided to search for different side channels leading to more practical oracles. As pointed out by James Manger on the official JOSE (JSON Object Signing and Encryption) mailing list,[8] an additional side channel could arise from an improper `Exception` handling in Java's PKCS#1 implementation.

[8] `http://www.ietf.org/mail-archive/web/jose/current/msg01936.html`

Side Channel Analysis. The Java PKCS#1 implementation strictly checks the message format according to Equation 1. The message must start with `0x00 02`, contain at least eight non-zero padding bytes, and a `0x00` byte indicating the end of the padding string. If this format is correct, the secret is extracted. Otherwise, a `BadPaddingException` is thrown. The method code can be found in Listing 9.

```
1  /**
2   * PKCS#1 v1.5 unpadding (blocktype 1 and 2).
3   */
4   private byte[] unpadV15(byte[] padded)
5    throws BadPaddingException {
6      int k = 0;
7      if (padded[k++] != 0) {
8        throw new BadPaddingException(
9         "Data must start with zero");
10     }
11     if (padded[k++] != type) {
12       throw new BadPaddingException(
13        "Blocktype mismatch: " + padded[1]);
14     }
15     while (true) {
16       int b = padded[k++] & 0xff;
17       if (b == 0) {
18         break;
19       }
20       if (k == padded.length) {
21         throw new BadPaddingException(
22          "Padding string not terminated");
23       }
24       if ((type == PAD_BLOCKTYPE_1)
25        && (b != 0xff)) {
26         throw new BadPaddingException(
27          "Padding byte not 0xff: " + b);
28       }
29     }
30     int n = padded.length - k;
31     if (n > maxDataSize) {
32       throw new BadPaddingException(
33        "Padding string too short");
34     }
35     byte[] data = new byte[n];
36     System.arraycopy(padded,
37      padded.length - n, data, 0, n);
38     return data;
39   }
```

Figure 9: Java's PKCS# v1.5 method for format check and padding removal can throw a BadPaddingException - *Source: sun.security.rsa.RSAPadding*

With our T.I.M.E. framework we investigated the JSSE server implementation which internally uses the PKCS#1 unpadding method described above. We sent PKCS#1 compliant and non-PKCS#1 compliant messages to the JSSE server and found that with non-PKCS#1 compliant messages an *additional* `Exception` could be provoked. The `Exception` was correctly handled by the JSSE logic and did not result in a distinguish-

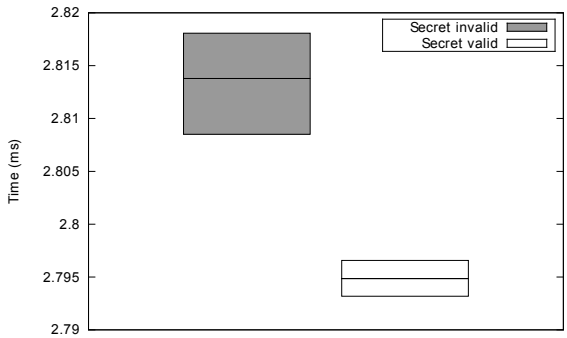


Figure 10: Timing measurement results for Java 1.7 (JSSE). The valid secret refers to a PKCS#1 compliant ciphertext. The invalid secret refers to a non-PKCS#1 compliant ciphertext. In the non-PKCS#1 compliant structure the first byte (which should be 0x00) was altered to 0x08 to provoke an exception on the TLS server.

able error message. Thus, it did not help to create a direct PKCS#1 validation oracle. However, `Exception` handling in Java (as well as in other object oriented languages) can introduce timing delays and thus slow down the whole application. Throwing, catching, and handling an `Exception` are time consuming tasks and thus lead to additional processing time.

Oracle Strength. *Timing Reliance.* We analyzed the timing differences between processing PKCS#1 compliant and non-PKCS#1 compliant messages on TLS servers running on Java 1.6 and 1.7 platforms. Figure 10 shows the filtered results of our time measurement with 5,000 queries. The results show differences of about 20 microseconds.

Probability Analysis. This behavior allows us to construct a new timing oracle:

$$\mathcal{O}_{T-exc}(c) = \begin{cases} 1 & \text{PKCS\#1 compliant} \\ 0 & \text{non-PKCS\#1 compliant (with an additional internal exception handling)} \end{cases}$$

$\mathcal{O}_{T-exc}$ is very permissive and much stronger than $\mathcal{O}_{T-rand}$, because it contains fewer plaintext validity checks. When working with 2048-bit keys, this oracle responds to a request starting with `0x00 02` with 1 with the following probability:

$$P^{2048}_{T-exc}(1|A) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{246}\right) \approx 0.6$$

Applying such an oracle results in much lesser queries and can thus be expected to be used for a practical attack.

Attack Evaluation. We used this timing oracle $\mathcal{O}_{T-exc}$ to perform a real Bleichenbacher attack in a switched LAN and proved the practicability of $\mathcal{O}_{T-exc}$. The attack on OpenJDK 1.6 took about 19.5 hours and 18,600 oracle queries.[9] About 20% of PKCS#1 compliant messages were identified as non-PKCS#1 compliant. The attack on Java 1.7 took about 55 hours and 20,662 queries. The larger number of queries and the longer processing time are caused by a higher value of false negatives (about 50%). The oracle identified about 467 PKCS#1 compliant messages incorrectly.

Mitigation. The object oriented architecture and especially the `Exception` handling of the JSSE implementation makes fixing the timing leak challenging. A common implementation pattern for RSA decryption is to provide a (generic) function to which the ciphertext is passed which returns the plaintext on success or an `Exception` otherwise. As stated, the generation of the `Exception` creates a detectable timing difference. Preparing an `Exception` at function entry, but not throwing it, leads to a smaller time difference, but might still be exploitable.

As a consequence we implemented a time constant PKCS#1 processing for SSL/TLS and proposed it as a fix for this issue to Oracle. The bug was assigned CVE-2014-411 and it was fixed with the *Oracle Java SE Critical Patch January 2014 – Java SE 7, Update 45* (and with the previous versions *Java SE 5u55* and *6u65*).

We verified that a similar timing behavior based on an additional exception is observable in a widespread BouncyCastle library.[10] BouncyCastle is implemented in two languages: Java and C#. We tested both implementations and locally invoked BouncyCastle PKCS#1 decryption methods. We could observe timing differences of about 20 microseconds between valid and invalid PKCS#1 messages in the Java and C# BouncyCastle version. This proved that the described timing behavior is not Java specific, and can be found in other object-oriented languages as well. We developed a patch for the Java version of BouncyCastle. We contacted the BouncyCastle developers with the proposed patch in March 2014.

8 Fourth Side Channel: Unexpected Timing Behavior by Hardware Appliances

The performance and practicability of the previous attacks motivated us to analyze further TLS stacks. We

[9] One oracle query is not equal to one server request. In order to respond to an oracle query, $\mathcal{O}_{T-exc}$ issued in our scenario up to 750 real server requests. It evaluated the response times and decided if the ciphertext was valid or not. See Figure 3.

[10] `https://www.bouncycastle.org`

had a chance to evaluate the behavior of *F5 BIG-IP* and *IBM Datapower* which use the *Cavium NITROX SSL accelerator chip*. Automated evaluation with our T.I.M.E. framework revealed that it was possible to execute a *complete* handshake, even though the encoded `PreMasterSecret` was of an incorrect format. More precisely, *F5 BIG-IP* and *IBM Datapower* did not verify the first byte of the PKCS#1 message and accepted messages which started with `0x??02` (where `0x??` represents an arbitrary byte).

Side Channel Analysis. This behavior does not lead to a direct attack. In order to correctly complete a handshake flow and receive a *Server*`Finished` message, an authenticated *Client*`Finished` message has to be sent to the server. Otherwise, the analyzed server responds with a `HANDSHAKE_FAILURE` message. Since the Bleichenbacher attacker is not in possession of the `PreMasterSecret`, he is not able to authenticate the *Client*`Finished` message and thus cannot trigger different messages. However, the server behavior strongly indicated that there could be a leakage in the PKCS#1 processing. Even though this leakage did not lead to different server responses, we assumed we could observe timing differences.

In comparison to the analysis described in the previous sections, we had no chance to review the code, because it is not publicly available. This turned our work to a black-box analysis and made it much harder.

Oracle Strength. We had a chance to evaluate the timing behavior of an *IBM Datapower* directly in our lab. The measurement machine was connected with a router to the *IBM Datapower* appliance.[11] We created different TLS requests based on our methodology (TLS compliant requests, PKCS#1 compliant requests, invalid requests etc.), and sent these requests to the server while the measurement machine observed the response times. The response times were finally compared using the NetTimer library.

The comparison of the response times confirmed our predictions and we could see clear timing differences by processing our TLS requests. The most visible timing difference was produced by requests starting with `0x??02`, see Figure 11. Based on this timing difference, the server behavior allowed to construct a new timing oracle:

$$\mathcal{O}_{T-hard} = \begin{cases} 1 & \text{starts with } \texttt{0x??02} \\ 0 & \textit{otherwise} \end{cases}$$

[11] In comparison to the previous measurements, the router did not route real traffic so our experiments were executed in a "lab" scenario.

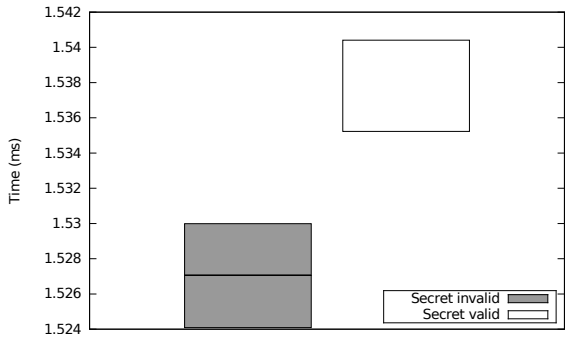


Figure 11: Timing measurement results for our *IBM Datapower*. The valid secret refers to a message, which starts with `0x??02`, where `0x??` indicates an arbitrary byte. The invalid secret refers to a message starting with different bytes.

However, this oracle is not compliant to the oracle used by Bleichenbacher. It responds with 1 to the request starting with `0x0102`, `0x0202`, `0x0302`, ... Thus, we needed to modify and adapt the original algorithm to handle this special case. This novel variant is described in Section 9.

Attack Evaluation. We evaluated the performance of our algorithm using a test oracle behaving like $\mathcal{O}_{T-hard}$. We repeated our experiment 500 times, with a 2048 bit RSA key. We needed about 4700 queries (median) to decrypt a ciphertext. This high performance is caused by the higher number of intervals the oracle accepts. Manger's attack [19] also reveals similar behavior.

We used the constructed timing oracle $\mathcal{O}_{T-hard}$ to perform a real attack on an *IBM Datapower* appliance. Our attacker needed 7371 oracle queries. The oracle correctly evaluated 2033 valid ciphertexts, while 1290 valid ciphertexts were incorrectly evaluated as invalid. The attack lasted 41 hours. The timing oracle $\mathcal{O}_{T-hard}$ issued about 4,000,000 server queries in total.

Mitigation. We communicated our findings to the vendors in November 2013. The current state of these issues can be tracked on their websites. F5 tracks this problem in their Bugzilla database under ID 435652. IBM gives their customers information about the current state in the Security Bulletin: SSL/TLS side channel attack on WebSphere DataPower (CVE-2014-0852).[12]

Since the Cavium products are used by other vendors like Cisco, Citrix or Juniper Networks, we assume that many other products were vulnerable, too.[13]

[12] `http://www.ibm.com/support/docview.wss?uid=swg21678204`

[13] `http://www.cavium.com/winning_products.html`

9 Novel Bleichenbacher Attack Variant

In the previous section we described a new oracle $\mathcal{O}_{T-hard}$. The oracle responds with 1 if a decrypted message starts with `0x??02`, where `0x??` represents an arbitrary byte. Such an oracle is not strong enough to implement Bleichenbacher's attack. The original algorithm from [5] is not able to tolerate false positives, it requires an oracle responding with 1 *only if* the decrypted message starts with `0x0002`. Note that $\mathcal{O}_{T-hard}$ is much weaker, as it responds with 1 if the message starts with `0x??02`. In the following we describe a novel variant of Bleichenbacher's attack, which is more robust than the original one and works also with the weaker oracle $\mathcal{O}_{T-hard}$.

We assume that the original message is PKCS#1 compliant and lies in the interval $[2B, 3B)$, where $B = 2^{8(\ell-2)}$. In this case the Bleichenbacher algorithm sets the starting interval containing the message of interest $m_0 \in [a,b]$, where $a = 2B$ and $b = 3B$

In the first step, the original algorithm searches for values $s > (2B+N)/3B$ such that $c = (c_0 \cdot s^e) \bmod N$ is decrypted to a PKCS#1 compliant message. This is not possible by applying $\mathcal{O}_{T-hard}$, since the oracle would respond with many false positives. We know that if $\mathcal{O}_{T-hard}$ responds with 1, the decrypted message starts with `0x0002`, `0x0102`, ... or `0xFF02`. This means the message lies in one of the following intervals: $[2B, 3B)$, $[258B, 259B)$, $[514B, 515B)$, ... If we start the algorithm with a large s value, we can easily produce a message from one of those intervals.

The basic idea behind our algorithm is to use the additional intervals and make the search more fine-grained. For this purpose, we define q, where $q \in \{1 \ldots (N/256B)\}$. In the first step, we set $r_0 = 0$ and iteratively search s_{ij} values by setting $q_j = 1 \ldots (N/256B)$:

$$\frac{2B + r_i N + q_j(256B)}{b} \leq s_{ij} < \frac{3B + r_i N + q_j(256B)}{a}.$$

We send $(c_0 \cdot s_{ij}^e) \bmod N$ to the server and observe its response. With each valid request, we can reduce the interval, where the original plaintext m_0 lies in:

$$a = max\left(a, \frac{2B + r_i N + q_j(256B)}{s_{ij}}\right)$$

$$b = min\left(b, \frac{3B + r_i N + q_j(256B)}{s_{ij}}\right)$$

Afterwards, we increment r and repeat the same steps for $q = 1 \ldots (N/256B)$.

The algorithm repeats these steps and reduces the possible solutions for m_0, until only one solution is left.

10 Other TLS Stacks

During our research we also analyzed other SSL/TLS implementations. *Microsoft Schannel* (Secure Channel) revealed no significant timing differences and behaves quite differently to any other stack: In case of processing errors of any kind, the connection is immediately terminated instead of sending alert messages. The timing measurements were too noisy to distill boundaries for distinguishing different processing branches. Due to the fact that the product is closed-source a code analysis was not possible.

11 Related Work

In this section we give a short overview on scientific publications analyzing side channel attacks and security of SSL/TLS. For a comprehensive list of SSL/TLS attacks we refer to [21].

Bleichenbacher Attacks. After publication of the original attack [5], several variants were discovered. Klima et al. found out that a strict verification of the TLS version number in the `PreMasterSecret` can lead to a side channel enabling Bleichenbacher's attack [16]. In [4] Bardou, Focardi, Kawamoto, Simionato, Steel and Tsay significantly improved Bleichenbacher's attack, and applied it to other PKCS#1-based environments.

Although Daniel Bleichenbacher conjectured that there might be timing-based side channels for Bleichenbacher attacks, they were discovered only for other protocols. For example, Jager et al. [13] describe a practical timing-based Bleichenbacher attack against implementations of the XML Encryption standard. They were able to exploit this side channel over a very noisy network (Planetlab) which was possible because timing differences could be increased by the attacker. During their research, they measured timing differences in the order of milliseconds whereas we had to cope with microseconds.

Timing Attacks on SSL/TLS. In 2003, Brumley and Boneh described an attack based on a timing side channel SSL/TLS [7], applicable if RSA is used for key exchange. Based on timing differences during processing of specially crafted `ClientKeyExchange` messages the private key of a server could successfully be extracted. Additionally, in 2011 Brumley and Tuveri [6] successfully attacked ECDSA based TLS connections (only OpenSSL stacks) by exploiting performance tweaks of the implementation.

Recent Attacks on SSL/TLS. The BEAST attack by Rizzo and Duong exploits predictable initialization vectors used by AES-CBC in TLS 1.0 [24]. The CRIME attack of the same authors shows that application of a compression method on plaintexts transported over SSL/-TLS can lead to serious practical attacks. Both attacks were theoretically discussed before [3, 15]. The authors showed how to apply them practically in specific scenarios by exploiting additional side channels. AlFardan and Paterson presented the Lucky13 padding oracle attack on AES-CBC [2] which exploits timing differences revealed by the HMAC computation over the decrypted data.

To practically deploy these attacks, a strong attacker is needed who is able to force the victim to *repeatedly* send the *same* data to the server. In contrast, our attacks exploit new side channels to mount Bleichenbacher's attack which enables to decrypt the whole `PreMasterSecret` (and thus the whole SSL/TLS session) without the need to control the user's client software.

Theoretical Results on TLS Security. After publication of Bleichenbacher's paper, the security of encoding schemes for RSA-based TLS was discussed intensively. However, due to the fact that the `Finished` messages are sent encrypted, no full security proof for TLS was available prior to 2012. In [12], a new security model (ACCE) was introduced by Jager et al., and a full proof for TLS-DHE with mutual authentication was given.

One year later, Krawczyk et al. gave a proof for the two remaining families of ciphersuites, TLS-RSA and TLS-DH, and for server-only authentication [17]. They prove security against Bleichenbacher attacks by proposing the following countermeasure: The server should use the `ClientFinished` message as a Message Authentication Code (MAC) for the `ClientKeyExchange` message. Only if `ClientFinished` is verified successfully, the server should continue the handshake by making further computations.

These two papers contain extensive related work sections, where all previous theoretical publications on TLS can be found. Theoretical security proofs must be treated carefully: The results can only be applied to practical implementations if all preconditions are satisfied, and if all cryptographic building blocks are implemented in an ideal way (i.e. yielding no side channels). Our results thus do not contradict the proofs, but simply show that the implementations of the building blocks are not ideal.

12 Future Work

TLS for non-HTTP protocols. The search for new error or timing-based side channels can be broadened to cover cryptographic protocol implementations in other application scenarios. Especially, protocols that use parts or concepts of SSL/TLS, such as EAP-TLS [1] or SSL/-TLS stacks of other languages and frameworks provide space for further investigation.

```
1  /**
2   * PKCS#1 v2.1 OAEP unpadding (MGF1).
3   */
4   private byte[] unpadOAEP(byte[] padded)
5    throws BadPaddingException {
6      byte[] EM = padded;
7      int hLen = lHash.length;
8
9      if (EM[0] != 0) {
10         throw new BadPaddingException(
11           "Data must start with zero");
12     }
13     ...
```

Figure 12: OAEP unpadding function of Java 7.

OAEP Comes to the Rescue. Many problems related to the *old* PKCS#1 are supposed to disappear with the introduction of OAEP [14]. However, during our research we also found problems in Java's OAEP processing. Listing 12 shows the code of Java's `RSAPadding.java` class which contains the logic for OAEP processing.

Line 9-12 outline a conditional branch that could be used to apply Manger's attack [19]. Patching is required. This example shows that OAEP is only of help if implemented correctly, i.e. without side channels.

We notified Oracle about this issue. The code was patched in the Java release from April 2014.

13 Conclusion

The problem of side channels leaking partial information about cryptographic computations seems to be much more persistent than expected: Error messages from standard libraries, and especially timig issues make generic solutions impossible.

The results of this paper show that Bleichenbacher attacks can still be used to break SSL/TLS implementations. Timing side channels underline the need for cryptographic libraries with branch independent, nearly time constant execution paths. The uncovered side channels motivate for the development of cryptographic penetration testing tools, able to detect such implementation deficiencies in the development phase.

Our results are alarming, especially when considering that Bleichenbacher attacks are known for about 15 years. They also show that PKCS#1 compliance checking is of prime importance to the security of a TLS implementations: Strict checks on TLS-PKCS#1 compli-

ance as performed by OpenSSL prevent Bleichenbacher attacks, even if side channels are present.

The question whether the introduction of RSA-OAEP padding would solve the problem still remains open: Only if RSA-OAEP is implemented without any side channels, the cryptographic features of this padding scheme can be enforced.

Acknowledgements

We would like to thank Graham Steel for providing us their improved Bleichenbacher attack code [4], and the security teams of Oracle, Cavium, IBM and F5 for their cooperation.

Furthermore, we would like to thank Tibor Jager, Christian Mainka, James Manger, and anonymous reviewers for their comments.

References

[1] ABOBA, B., BLUNK, L., VOLLBRECHT, J., CARLSON, J., AND LEVKOWETZ, H. Extensible Authentication Protocol (EAP). RFC 3748 (Proposed Standard), June 2004. Updated by RFC 5247.

[2] ALFARDAN, N. J., AND PATERSON, K. G. Lucky thirteen: Breaking the tls and dtls record protocols. *2013 IEEE Symposium on Security and Privacy 0* (2013), 526–540. `http://www.isg.rhul.ac.uk/tls/TLStiming.pdf`.

[3] BARD, G. V. The vulnerability of ssl to chosen plaintext attack. *IACR Cryptology ePrint Archive 2004* (May 2004), 111.

[4] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., STEEL, G., AND TSAY, J.-K. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO* (2012), Canetti and R. Safavi-Naini, Eds.

[5] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.

[6] BRUMLEY, B., AND TUVERI, N. Remote Timing Attacks Are Still Practical. In *Computer Security - ESORICS 2011*, vol. 6879 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Sept. 2011.

[7] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (June 2003), SSYM'03, USENIX Association.

[8] CROSBY, S. A., WALLACH, D. S., AND RIEDI, R. H. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur. 12*, 3 (Jan. 2009), 17:1–17:29.

[9] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746.

[10] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746.

[11] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFC 5746.

[12] JAGER, T., KOHLAR, F., SCHÄGE, S., AND SCHWENK, J. On the security of tls-dhe in the standard model. In *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 273–293.

[13] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's attack strikes again: Breaking pkcs#1 v1.5 in xml encryption. In *ESORICS* (2012), S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*, Springer, pp. 752–769.

[14] JONSSON, J., AND KALISKI, B. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.

[15] KELSEY, J. Compression and information leakage of plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers* (Nov. 2002), vol. 2365 of *Lecture Notes in Computer Science*, Springer.

[16] KLÍMA, V., POKORNÝ, O., AND ROSA, T. Attacking RSA-Based Sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, vol. 2779 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Sept. 2003.

[17] KRAWCZYK, H., PATERSON, K. G., AND WEE, H. On the Security of the TLS Protocol: A Systematic Analysis. Cryptology ePrint Archive, Report 2013/339, 2013. `http://eprint.iacr.org/`.

[18] M. D. GREEN (@OPENSSLFACT). OpenSSL vs. best practices (RSA decryption edition). 2.10.2012. 16:04, Tweet, `https://twitter.com/OpenSSLFact/status/253060773218222081`.

[19] MANGER, J. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs #1 v2.0. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings* (2001), vol. 2139 of *Lecture Notes in Computer Science*, Springer, pp. 230–238.

[20] MEYER, C. *20 Years of SSL/TLS Research : An Analysis of the Internet's Security Foundation*. PhD thesis, Ruhr-University Bochum, Feb. 2014.

[21] MEYER, C., AND SCHWENK, J. SoK: Lessons Learned From SSL/TLS Attacks. In *Proceedings of the 14th International Workshop on Information Security Applications* (Berlin, Heidelberg, Aug. 2013), WISA 2013, Springer-Verlag.

[22] PATERSON, K. G., RISTENPART, T., AND SHRIMPTON, T. Tag size does matter: attacks and proofs for the TLS record protocol. In *Proceedings of the 17th international conference on The Theory and Application of Cryptology and Information Security* (Dec. 2011), ASIACRYPT'11, Springer-Verlag.

[23] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[24] RIZZO, J., AND DUONG, T. Here Come The XOR Ninjas, May 2011.

Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns

Jonathan Dautrich
University of California, Riverside

Emil Stefanov
University of California, Berkeley

Elaine Shi
University of Maryland, College Park

Abstract

We present Burst ORAM, the first oblivious cloud storage system to achieve both practical response times and low total bandwidth consumption for bursty workloads. For real-world workloads, Burst ORAM can attain response times that are nearly optimal and orders of magnitude lower than the best existing ORAM systems by reducing online bandwidth costs and aggressively rescheduling shuffling work to delay the bulk of the IO until idle periods.

We evaluate our design on an enterprise file system trace with about 7,500 clients over a 15 day period, comparing to an insecure baseline encrypted block store without ORAM. We show that when baseline response times are low, Burst ORAM response times are comparably low. In a 32TB ORAM with 50ms network latency and sufficient bandwidth capacity to ensure 90% of requests have baseline response times under 53ms, 90% of Burst ORAM requests have response times under 63ms, while requiring only 30 times the total bandwidth consumption of the insecure baseline. Similarly, with sufficient bandwidth to ensure 99.9% of requests have baseline responses under 70ms, 99.9% of Burst ORAM requests have response times under 76ms.

1 Introduction

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns hinder its growth [3]. Encryption alone is insufficient to ensure privacy in storage outsourcing applications, as information about the contents of encrypted records may still leak via data access patterns. Existing work has shown that access patterns on an encrypted email repository may leak sensitive keyword search queries [12], and that accesses to encrypted database tuples may reveal ordering information [5].

Oblivious RAM (ORAM), first proposed in a groundbreaking work by Goldreich and Ostrovsky [8, 9], is a cryptographic protocol that allows a client to provably hide access patterns from an untrusted storage server. Recently, the research community has focused on making ORAM schemes practical for real-world applications [7, 11, 21, 23–25, 27]. Unfortunately, even with recent improvements, ORAMs still incur substantial bandwidth and response time costs.

Many prior ORAM works focus on minimizing bandwidth consumption. Several recent works on cloud-based ORAMs achieve low bandwidth costs using a large amount of client-side storage [11, 23, 24]. Others rely on expensive primitives like PIR [17] or additional assumptions such as trusted hardware [15] or non-colluding servers [22] to reduce bandwidth costs.

To be practical, ORAM must also minimize response times observed by clients for each request. We propose Burst ORAM, a novel ORAM that dramatically reduces response times for realistic workloads with bursty characteristics. Burst ORAM is based on ObliviStore [23], the most bandwidth-efficient existing ORAM.

Burst ORAM uses novel techniques to minimize the *online* work of serving requests and delay *offline* block shuffling until idle periods. Under realistic bursty loads, Burst ORAM achieves orders of magnitude shorter response times than existing ORAMs, while retaining total bandwidth costs less than 50% higher than ObliviStore's.

During long bursts, Burst ORAM's behavior automatically and gracefully degrades to be similar to that of ObliviStore. Thus, even in a worst-case workload, Burst ORAM's response times and bandwidth costs are competitive with those of existing ORAMs.

We simulate Burst ORAM on a real-world corporate data access workload (7,500 clients and 15 days) to show that it can be used practically in a corporate cloud storage environment. We compare against an insecure baseline encrypted block store without ORAM and show that when baseline response times are low, Burst ORAM response times are also low. In a 32TB ORAM with 50ms network latency and sufficient bandwidth capacity to ensure 90% of requests have baseline response times un-

der 53ms, 90% of Burst ORAM requests have response times under 63ms. Similarly, with sufficient bandwidth to ensure 99.9% of requests have baseline responses under 70ms, 99.9% of Burst ORAM requests have response times under 76ms. Existing works exhibit response times on the order of seconds or higher, due to high bandwidth [11, 23, 25, 28] or computation [17] requirements. To our knowledge, our work is the first to evaluate ORAM response times on a realistic, bursty workload.

As in previous ORAM schemes, we do not seek to hide the timing of data requests. Thus, we assume request start times and durations are known. To ensure security, we do not allow the IO scheduler to make use of the data access sequence or other sensitive information. We analyze Burst ORAM security in Section 6.4.

1.1 Burst ORAM Contributions

Burst ORAM introduces several techniques for reducing response times and keeping bandwidth costs low that distinguish it from ObliviStore and other predecessors.

Novel scheduling policies. Burst ORAM prioritizes the *online* work that must be complete before requests are satisfied. If possible, our scheduler delays shuffle work until off-peak times. Delaying shuffle work consumes client-side storage, so if a burst is sufficiently long, client space will fill, forcing shuffling to resume. By this time, there are typically multiple shuffle jobs pending.

We use a greedy strategy to prioritize jobs that free the most client-side space per unit of shuffling bandwidth consumed. This strategy allows us to sustain lower response times for longer during an extended burst.

Reduced online bandwidth costs. We propose a new *XOR technique* that reduces the online bandwidth cost from $O(\log N)$ blocks per request in ObliviStore to nearly 1, where N is the outsourced block count. The XOR technique can also be applied to other ORAM implementations such as SR-ORAM [26] (see Appendix B).

Level caching. We propose a new technique for using additional available client space to store small levels from each partition. By caching these levels on the client, we are able to reduce total bandwidth cost substantially.

1.2 Related Work

Oblivious RAM was first proposed in a seminal work by Goldreich and Ostrovsky [9]. Since then, a fair amount of theoretic work has focused on improving its asymptotic performance [1, 4, 10, 11, 13, 18, 19, 21, 24, 27]. Recently, there has been much work designing and optimizing ORAM for cloud-based storage outsourcing settings, as noted below. Different ORAMs provide varying trade-offs between bandwidth cost, client/server storage, round complexity, and computation.

ORAM has been shown to be feasible for secure (co-)processor prototypes, which prevent information leakage due to physical tampering [6, 15, 16, 20]. Since on-chip trusted cache is expensive, such ORAM schemes need constant or logarithmic client-side storage, such as the binary-tree ORAM [21] and its variants [7, 17, 25].

In cloud-based ORAMs, the client typically has more space, capable of storing $O(\sqrt{N})$ blocks or a small amount of per-block metadata [10, 23, 24, 28] that can be used to reduce ORAM bandwidth costs. Burst ORAM also makes such client space assumptions.

Online and offline costs for ORAM were first made explicit by Boneh et al. [1] They propose a construction that has $O(1)$ online but $O(\sqrt{N})$ overall bandwidth cost. The recent Path-PIR work by Mayberry et al. [17] mixes ORAM and PIR to achieve $O(1)$ online bandwidth cost with an overall bandwidth cost of $O(\log^2 N)$ with constant client memory. Unfortunately, the PIR is still computationally expensive, so their scheme requires more than 40 seconds for a read from a 1TB database [17]. Burst ORAM has $O(1)$ online and $O(\log N)$ overall bandwidth cost, without the added overhead of PIR.

Other ORAMs that do not rely on trusted hardware or non-colluding servers have $\Omega(\log N)$ online bandwidth cost including works by Williams, Sion, et al. [27, 28]; by Goodrich, Mitzenmacher, Ohrimenko, and Tamassia [10, 11]; by Kushilevitz et al. [13]; and by Stefonov, Shi, et al. [21, 23–25]. Burst ORAM handles bursts much more effectively by reducing the online cost to nearly 1 block transfer per block request during a burst, greatly reducing response times.

2 Preliminaries

2.1 Bandwidth Costs

Bandwidth consumption is the primary cost in many modern ORAMs, so it is important to define how we measure its different aspects. Each block transferred between the client and server is a single unit of IO. We assume that blocks are large in practice (at least 1KB), so transferred meta-data (block IDs) have negligible size.

Definition 1 *The* bandwidth cost *of a storage scheme is given by the average number of blocks transferred in order to read or write a single block.*

We identify bandwidth costs by appending X to the number. A bandwidth cost of 2X indicates two blocks transferred per request, which is twice the cost of an unprotected scheme. We consider *online*, *offline*, *effective*, and *overall* IO and bandwidth costs, where each cost is given by the average amount of the corresponding type of IO.

Online IO consists of the block transfers needed before a request can be safely marked as satisfied, assuming the scheme starts with no pending IO. The *online bandwidth cost* of a storage scheme without ORAM is just 1X — the

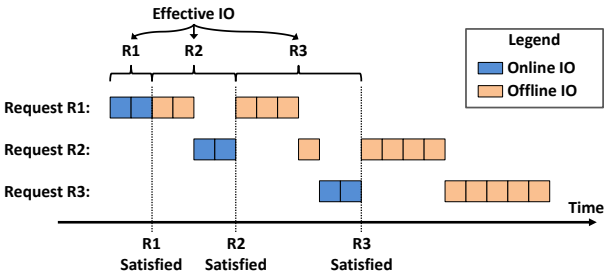


Figure 1: Simplified scheme with sequential IO and contrived capacity for delaying offline IO. 3 requests require same online (2), offline (5), and overall (7) IO. Online IO for R1 is handled immediately, so R1's effective IO is only 2. R2 waits for 2 units of offline IO from R1, so its effective IO is 4. R3 waits for the rest of R1's offline IO, plus one unit of R2's offline IO, so its effective IO is 6.

IO cost of downloading the desired block. In ORAM it may be higher, as additional blocks may be downloaded to hide the requested block's identity.

Offline IO consists of transfers needed to prepare for subsequent requests, but which may be performed after the request is satisfied. Without ORAM, the *offline bandwidth cost* is 0X. In ORAM it is generally higher, as additional *shuffle IO* is needed to obliviously permute blocks in order to guarantee privacy for future requests.

Overall IO / bandwidth cost is just the sum of the online and offline IO / bandwidth costs, respectively.

Effective IO consists of all online IO plus any pending offline IO from previous requests that must be issued before the next request can be satisfied. Without ORAM, effective IO and online IO are equal. In traditional ORAMs, offline IO is issued immediately after each request's online IO, so effective and overall IO are equal. In Burst ORAM, we delay some offline IO, reducing each request's effective IO as illustrated in Figure 1. Smaller effective costs mean less IO between requests, and ultimately shorter response times.

ORAM reads and writes are indistinguishable, so writes have the same bandwidth costs as reads.

2.2 Response Time

The *response time* of a block request (ORAM read/write operation) is defined as the lapse of wall-clock time between when the request is first issued by the client and when the client receives a response. The *minimum* response time is the time needed to perform all online IO. Response times increase when offline IO is needed between requests, increasing effective IO, or when requests are issued rapidly in a burst, delaying later requests.

2.3 ObliviStore ORAM

Burst ORAM builds on ObliviStore [23], so we give an overview of the scheme here. A full description of the ObliviStore system and its ORAM algorithm spans about 55 pages [23, 24], so we describe it at a high level, focusing only on components relevant to Burst ORAM.

Partitions and levels. ObliviStore stores N logical data blocks. Each block is encrypted using a standard symmetric key encryption scheme before it is stored on the server. Every time a block is uploaded by the client, it is re-encrypted using a new nonce to prevent linking.

ObliviStore securely splits blocks into $O(\sqrt{N})$ *partitions* of $O(\sqrt{N})$ blocks each. Each partition is an ORAM consisting of $O(\log N)$ *levels* with $2,4,8,\ldots,O(\sqrt{N})$ blocks each. Newly created levels are filled with half encrypted real blocks and half encrypted dummies, randomly permuted so that reals and dummies are indistinguishable to the server. Each level is occupied only half the time on average. The client has space to store $O(\sqrt{N})$ blocks and the locations of all N blocks.

Requests. When the client makes a block request, whether a read or write, the block must first be downloaded from the appropriate partition. To maintain obliviousness, ObliviStore must fetch one block from every non-empty level in the target partition ($O(\log N)$ blocks of *online* IO). Only one fetched block is real, and the rest are dummies, except in the case of early shuffle reads described below. Once a dummy is fetched, it is discarded, and new dummies are created later as needed. ObliviStore securely processes multiple requests in parallel, enabling full utilization of available bandwidth capacity.

Eviction. Once the real block is fetched, it is updated or returned to the client as necessary, then ranodmly assigned to a new partition p. The block is not immediately uploaded, but is scheduled for *eviction* to p and stored in a client-side *data cache*. An independent eviction process later obliviously evicts the block from the cache to p. The eviction triggers a write operation on p's ORAM, which creates or enlarges a *shuffling job* for p.

Shuffling Jobs. Each partition p has at most one pending *shuffle job*. A job consists of downloading up to $O(\sqrt{N})$ blocks from p, permuting them on the client with recent evictions and new dummies, and uploading. Shuffle jobs incur *offline* IO, and vary in *size* (amount of IO) from $O(1)$ to $O(\sqrt{N})$. Intuitively, to ensure that non-empty levels have at least one dummy left, we must re-shuffle a level once half its blocks have been removed. Larger levels need shuffling less often, so larger jobs occur less frequently, keeping offline bandwidth costs at $O(\log N)$.

Shuffle IO scheduling. A fixed amount of $O(\log N)$ shuffle IO is performed after each request to amortize the work required for large jobs. The IO for jobs from multi-

ple partitions may be executed in parallel: while waiting on reads for one partition, we may issue reads or writes for another. Jobs are started in the order they are created.

Early shuffle reads. *Early shuffle reads*, referred to as *early cache-ins* or *real cache-ins* in ObliviStore, occur when a request needs to fetch a block from a level, but at least half the level's original blocks have been removed. In this case, we cannot guarantee that any dummies remain. Thus, early shuffle reads must be treated as real blocks and stored separately by the client until they are returned to the server as part of a shuffle job. We call such reads *early shuffle reads* since the blocks would have eventually been read during a shuffle job. Early shuffle reads are infrequent, but made possible since ObliviStore performs requests while shuffling is in progress.

Level compression. ObliviStore uses a technique called *level compression* [24] to compress blocks uploaded during shuffling. It allows the client to upload k real and k dummy blocks using only k blocks of bandwidth without revealing which k are dummies. Level compression reduces only the offline (shuffling) bandwidth cost.

3 Overview of our Approach

Traditional ORAMs focus on reducing average and worst-case *overall bandwidth costs* (per-request overall IO). However, even the most bandwidth-efficient schemes [23, 24] suffer from a 20X–35X bandwidth cost.

In this paper, we instead focus on reducing *effective* IO by reducing online IO and delaying offline IO. We can then satisfy bursts of requests quickly, delaying most IO until idle periods. Figure 2 illustrates this concept.

Our approach allows many bursts to be satisfied with nearly a 1X effective bandwidth cost. That is, during the burst, we transfer just over one block for every block requested. After the burst we do extra IO to catch up on shuffling and prepare for future requests. Our approach maintains an overall bandwidth cost less than 50% higher than [23, 24] in practice (see Figure 12 in Section 7).

Bursts. Intuitively, a burst is a period of frequent block requests from the client preceded and followed by relatively idle periods. Many real-world workloads exhibit bursty patterns (e.g. [2, 14]). Often, bursts are not discrete events, such as when multiple network file system users are operating concurrently. Thus we handle bursts fluidly: the more requests issued at once, the more Burst ORAM tries to delay offline IO until idle periods.

Challenges. We are faced with two key challenges when building a burst-friendly ORAM system. The first is ensuring that we maintain security. A naive approach to reducing online IO may mark requests as satisfied before enough blocks are read from the server, leaking information about the requested block's identity.

The second challenge is ensuring that we maximally utilize client storage and available bandwidth while avoiding deadlock. An excessively aggressive strategy that delays too much IO may use so much client space that we run out of room to shuffle. It may also under-utilize available bandwidth, increasing response times. On the other hand, an overly conservative strategy may under-utilize client space or perform shuffling too early, delaying online IO and increasing response times.

Techniques and Outline. In Burst ORAM, we address these challenges by combining several novel techniques. In Section 4 we introduce our XOR technique for reducing online bandwidth cost to nearly 1X. We also describe our techniques for prioritizing online IO and delaying offline/shuffle IO until client memory is nearly full. In Section 5 we show how Burst ORAM prioritizes efficient shuffle jobs in order to delay the bulk of the shuffle IO even further, ensuring that we minimize effective IO during long bursts. We then introduce a technique for using available client space to cache small levels locally to reduce shuffle IO in both Burst ORAM and ObliviStore.

In Section 6 we discuss the system-level techniques used in Burst ORAM, and present its design in detail. In Section 7, we evaluate Burst ORAM's performance through micro-benchmarks and extensive simulations.

4 Prioritizing and Reducing Online IO

Existing ORAMs require high online and offline bandwidth costs to obscure access patterns. ObliviStore must fetch one block from every level in a partition (see Section 2.3), requiring $O(\log N)$ online IO per request. Figure 3 (left) illustrates this behavior. After each request, ObliviStore also requires $O(\log N)$ offline/shuffle IO. Since ObliviStore issues online and offline IO before satisfying the next request, its effective IO is high, leading to large response times during bursts. Other ORAMs work differently, such as Path ORAM [25] which organizes data as a tree, but still have high effective costs. We now show how Burst ORAM achieves lower effective bandwidth costs and response times than ObliviStore.

4.1 Prioritizing Online IO

One way we achieve low response times in Burst ORAM is by prioritizing online IO over shuffle IO. That is, we suppress shuffle IO during bursts, delaying it until idle periods. Requests are satisfied once online IO finishes,[1] so prioritizing online IO allows us to satisfy all requests before any shuffle IO starts, keeping response times low even for later requests. Figure 2 illustrates this behavior.

During the burst, we continue processing requests by fetching blocks from the server, but since shuffling is suppressed, no blocks are uploaded. Thus, we must resume shuffling once client storage fills. Section 5.2 dis-

[1] Each client write also incurs a read, so writes still incur online IO.

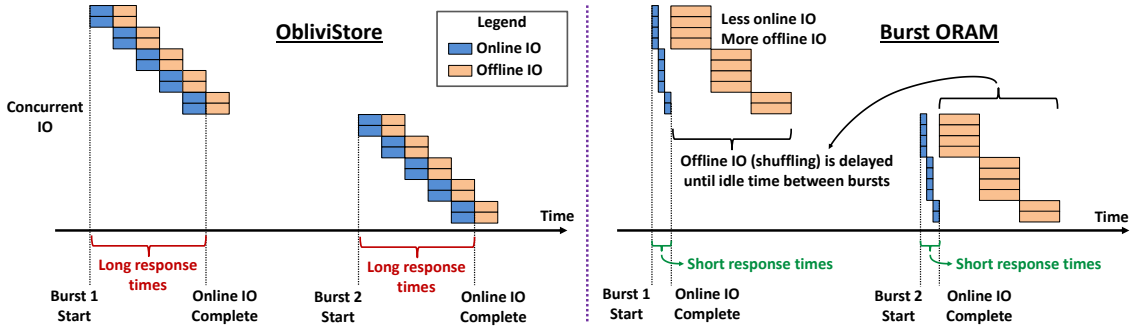


Figure 2: **Reducing response time.** Because Burst ORAM (right) does much less online IO than ObliviStore (left) and delays offline IO, it is able to respond to ORAM requests much faster. In this (overfly simplified) illustration, the bandwidth capacity is enough to transfer 4 blocks concurrently. Both ORAM systems do the same amount of IO.

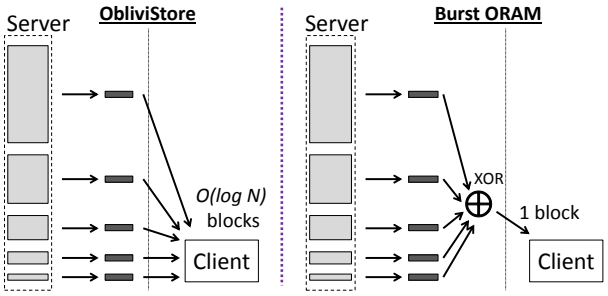


Figure 3: **Reducing online cost.** In ObliviStore (left) the online bandwidth cost is $O(\log N)$ blocks of IO on average. In Burst ORAM (right), we reduce online IO to only one block, improving handling of bursty traffic.

cusses how to delay shuffle IO even further. Section 6 details changes from the ObliviStore design required to avoid deadlock and fully utilize client space.

When available bandwidths are large and bursts are short, the response time saved by prioritizing online IO is limited, as most IO needed for the burst can be issued in parallel. However, when bandwidth is limited or bursts are long, the savings can be substantial. With shuffle IO delayed until idle times, online IO dominates the effective IO, becoming the bottleneck during bursts. Thus we can further reduce response times by reducing online IO.

4.2 XOR Technique: Reducing Online IO

We introduce a new mechanism called the *XOR technique* that allows the Burst ORAM server to combine the $O(\log N)$ blocks fetched during a request into a single block that is returned to the client (Figure 3 right), reducing the online bandwidth cost to $O(1)$.

If we fetched only the desired block, we would reveal its identity to the server. Instead, we XOR all the blocks together and return the result. Since there is at most one real block among the $O(\log N)$ returned, the client can locally reconstruct the dummy block values and XOR them with the returned block to recover the encrypted real block. XOR technique steps are shown in Figure 4.

4.2.1 XOR Technique Details

In Burst ORAM, as in ObliviStore, each request needs to retrieve a block from a single *partition*, which is a simplified hierarchical ORAM resembling those in [9]. The hierarchy contains $L \approx \frac{1}{2}\log_2 N$ levels with real-block capacities $1, 2, 4, \ldots, 2^{L-1}$ respectively.

To retrieve a requested block, the client must fetch exactly one block from each of the L levels. The XOR technique requires that the client be able to reconstruct dummy blocks, and that dummies remain indistinguishable from real blocks. We achieve this property by encrypting a real block b residing in partition p, level ℓ, and offset *off* as $\mathsf{AES}_{sk_{p,\ell}}(off||B)$. We encrypt a dummy block residing in partition p, level ℓ, and offset *off* as $\mathsf{AES}_{sk_{p,\ell}}(off)$. The key $sk_{p,\ell}$ is specific to partition p and level ℓ, and is randomized every time ℓ is rebuilt.

For simplicity, we start by considering the case without early shuffle reads. In this case, exactly one of the L blocks requested is the encryption of a real block, and the rest are encryptions of dummy blocks. The server XORs all L encrypted blocks together into a single block X_Q that it returns to the client. The client knows which blocks are dummies, and knows p, ℓ, off for each block, so it reconstructs all the encrypted dummy blocks and XORs them with X_Q to obtain the encrypted requested/real block.

4.2.2 Handling early shuffle reads

An early shuffle read occurs when we need to read from a level with no more than half its original blocks remaining. Since such early shuffle reads may be real blocks, they cannot be included in the XOR. Fortunately, the number of blocks in a level is public, so the server already knows which levels will cause early shuffle reads. Thus, the server simply returns early shuffle reads individually, then XORs the remaining blocks, leaking no information about the access sequence.

1. Client issues block requests to server, one per level
2. Server, to satisfy request
   (a) Retrieves and returns early shuffle reads
   (b) XORs remaining blocks together into single *combined* block and returns it
3. Client, while waiting for response
   (a) Regenerates encrypted dummy block for each non-early-shuffle-read
   (b) XORs all dummies to get *subtraction* block
4. Client receives combined block from server and XORs with subtraction block to get requested block
5. Client decrypts requested block

Figure 4: XOR Technique Steps

Since each early shuffle read block must be transferred individually, early shuffle reads increase online IO. Fortunately, early shuffle reads are rare, even while shuffling is suppressed during bursts, so the online bandwidth cost stays under 2X and near 1X in practice (see Figure 7).

4.2.3 Comparison with ObliviStore

ObliviStore uses *level compression* to reduce shuffle IO. When the client uploads a level to the server, it first compresses the level down to the combined size of the level's real blocks. Since half the blocks are dummies, half the upload shuffle IO is eliminated. For details on level compression and its security, see [24].

Unfortunately, Burst ORAM's XOR technique is incompatible with level compression due to discrepancies in the ways dummy blocks must be formed. The XOR technique requires that the client be able to reconstruct dummy blocks locally, so in Burst ORAM, each dummy's position determines its contents. In level compression, each level's dummy block contents are a function of the level's real block contents. Since the client cannot know the contents of all real blocks in the level, it cannot reconstruct the dummies locally.

Level compression and the XOR technique yield comparable overall IO reductions, though level compression performs slightly better. For example, the experiment in Figure 8 incurs roughly 23X and 26X overall bandwidth cost using level compression and XOR respectively. However, the XOR technique reduces *online* IO, while level compression reduces *offline* IO, so the XOR technique is more effective at reducing response times.

5 Scheduling and Reducing Shuffle IO

In Burst ORAM, once client space fills, we must start shuffling in order to return blocks to the server and continue the burst. If we are not careful about shuffle IO scheduling, we may immediately start doing large amounts of IO, dramatically increasing response times.

In this section, we show how Burst ORAM schedules shuffle IO so that jobs that free the most client space using the least shuffle IO are prioritized. Thus, at all times, Burst ORAM issues only the minimum amount of effective IO needed to continue the burst, keeping response times lower for longer. We also show how to reduce overall IO by locally caching the smallest levels from each partition. We start by defining *shuffle jobs*.

5.1 Shuffle Jobs

In Burst ORAM, as in ObliviStore, shuffle IO is divided into per-partition *shuffle jobs*. Each job represents the work needed to shuffle a partition p and upload blocks evicted to p. A shuffle job is defined by five entities:

- A partition p to which the job belongs
- Blocks evicted to but not yet returned to p
- Levels to read blocks from
- Levels to write blocks to
- Blocks already read from p (early shuffle reads)

Each shuffle job moves through three phases:

Creation Phase. We create a shuffle job for p when a block is evicted to p following a request. Every job starts out *inactive*, meaning we have not started work on it. If another block is evicted to p, we update the sets of eviction blocks and read/write levels in p's inactive job.

When Burst ORAM *activates* a job, it moves the job to the *Read Phase*, freezing the eviction blocks and read/write levels. Subsequent evictions to p will create a new *inactive* shuffle job. At any time, there is at most one active and one inactive shuffle job for each partition.

Read Phase. Once a shuffle job is activated, we begin fetching all blocks still on the server that need to be shuffled. That is, all previously unread blocks from all the job's read levels. Once all such blocks are fetched, they are *shuffled* with all blocks evicted to p and any early shuffle reads from the read levels. Shuffling consists of adding/removing dummies, pseudo-randomly permuting the blocks, and then re-encrypting each block. Once shuffling completes, we move the job to the *Write Phase*.

Write Phase. Once a job is shuffled we begin storing all shuffled blocks to the job's write levels on the server. Once all writes finish, the job is marked complete, and Burst ORAM is free to activate p's inactive job, if any.

5.2 Prioritizing Efficient Jobs

Since executing shuffle IO delays the online IO needed to satisfy requests, we can reduce response times by doing as little shuffling as is needed to free up space. The hope is that we can delay the bulk of the shuffling until an idle period, so that it does not interfere with pending requests.

By the time client space fills, there will be many partitions with inactive shuffle jobs. Since we can choose jobs in any order, we can minimize the up-front shuffling work by prioritizing the most *efficient* shuffle jobs: those

that free up the most client space per unit of shuffle IO. The space freed by completing a job for partition p is the number of blocks evicted to p plus the number of early shuffle reads from the job's read levels. Thus, we can define shuffle job efficiency as follows:

$$\text{Job Efficiency} = \frac{\#\text{ Evictions} + \#\text{ Early Shuffle Reads}}{\#\text{ Blocks to Read} + \#\text{ Blocks to Write}}$$

Job efficiencies vary substantially. Most jobs start with 1 eviction and 0 early shuffle reads, so their relative efficiencies are determined strictly by the sizes of the job's read and write levels. If the partition's bottom level is empty, no levels need be read, and only the bottom must be written, for an overall IO of 2 an an efficiency of 0.5. If instead the bottom 4 levels are occupied, all 4 levels must be read, and the 5th level written, for an total of roughly 15 reads and 32 writes, yielding a much lower efficiency of just over 0.02. Both jobs free equal amounts of space, but the higher-efficiency job uses less IO.

Since small levels are written more often than large ones, efficient jobs are common. Further, by delaying an unusually inefficient job, we give it time to accumulate more evictions. While such a job will also accumulate more IO, the added write levels are generally small, so the job's efficiency tends to improve with time. Thus, prioritizing efficient jobs reduces shuffle IO during the burst, thereby reducing response times.

Unlike Burst ORAM, ObliviStore does not use client space to delay shuffling, so there are fewer shuffle jobs to choose from at any one time. Thus, job scheduling is less important and jobs are chosen in creation order. Since ObliviStore is concerned with throughput, not response times, it has no incentive to prioritize efficient jobs.

5.3 Reducing Shuffle IO via Level Caching

Since small, efficient shuffle jobs are common, Burst ORAM spends a lot of time accessing small levels. If we use client space to locally cache the smallest levels of each partition, we can eliminate the shuffle IO associated with those levels entirely. Since levels are shuffled with a frequency inversely proportional to their size, each is responsible for roughly the same fraction of shuffle IO. Thus, we can greatly reduce shuffle IO by caching even a few levels from each partition. Further, since caching a level eliminates its early shuffle reads, which are common for small levels, caching can also reduce online IO.

We are therefore faced with a tradeoff between using client space to store requested blocks, which reduces response times for short bursts, and using it for local level caching, which reduces overall bandwidth cost.

5.3.1 Level Caching in Burst ORAM

In Burst ORAM, we take a conservative approach, and cache only as many levels as are guaranteed to fit in the worst case. More precisely, we identify the maximum number λ such that the client could store all real blocks from the smallest λ levels of every partition even if all were full simultaneously. We cache levels by only updating an inactive job when the number of evictions is such that all the job's write levels have index at least λ.

Since each level is only occupied half the time, caching λ levels consumes at most half of the client's space on average, leaving the rest for requested blocks. As we show experimentally in Section 7, level caching greatly reduces overall bandwidth cost, and can even reduce response times since it avoids early shuffle reads.

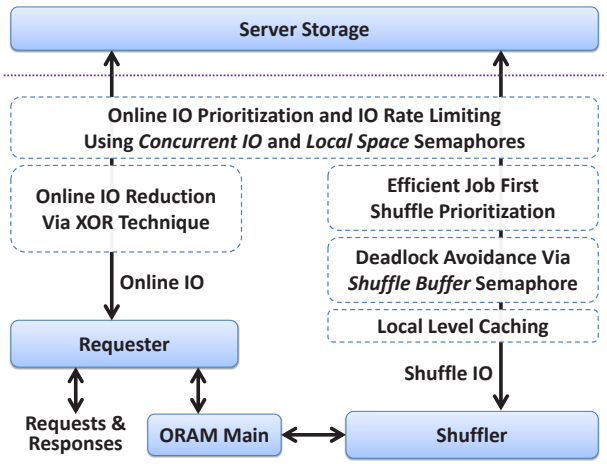


Figure 5: **Burst ORAM Architecture.** Solid boxes represent key system components, while dashed boxes represent functionality and the effects of the system on IO.

6 Detailed Burst ORAM Design

The Burst ORAM design is based on ObliviStore, but incorporates many fundamental functional and system-level changes. For example, Burst ORAM replaces or revises all the semaphores used in ObliviStore to achieve our distinct goal of online IO prioritization while maintaining security and avoiding deadlock. Burst ORAM also maximizes client space utilization, implements the XOR technique to reduce online IO, revises the shuffler to schedule efficient jobs first, and implements level caching to reduce overall IO.

6.1 Overall Architecture

Figure 5 presents the basic architecture of Burst ORAM, highlighting key components and functionality. Burst ORAM consists of two primary components, the online *Requester* and the offline *Shuffler*, which are controlled by the main event loop *ORAM Main*. Client-side memory allocation is shown in Figure 6.

ORAM Main accepts new block requests (reads and writes) from the client, and adds them to a *Request*

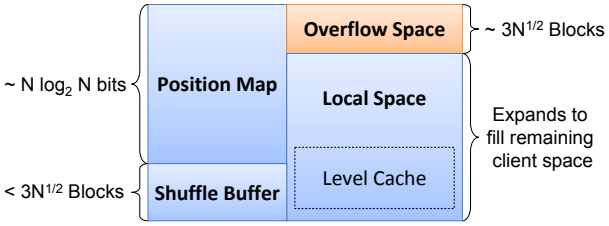


Figure 6: **Burst ORAM Client Space Allocation.** Fixed client space is reserved for the position map and shuffle buffer. A small amount of overflow space is needed for blocks assigned but not yet evicted (*data cache* in [24]). Remaining space is managed by *Local Space* and contains evictions, early shuffle reads, and the level cache.

Queue. On each iteration, ORAM Main tries advancing the Requester first, only advancing the Shuffler if the Requester needs no IO, thereby prioritizing online IO. The Requester and Shuffler use *semaphores* (Section 6.2) to regulate access to network bandwidth and client space.

The *Requester* reads each request from the Request Queue, identifies the desired block's partition, and fetches it along with any necessary dummies. To ensure oblivious behavior, the Requester must wait until all dummy blocks have been fetched before marking the request satisfied. All Requester IO is considered *online*.

The *Shuffler* re-encrypts blocks fetched by the Requester, shuffles them with other blocks, and returns them to the server. The Shuffler is responsible for managing shuffle jobs, including prioritizing efficient jobs and implementing level caching. All IO initiated by the shuffler is considered *offline* or *shuffle* IO.

6.2 Semaphores

Resources in Burst ORAM are managed via *semaphores*, as in ObliviStore. Semaphores are updated using only server-visible information, so ORAM can safely base its behavior on semaphores without revealing new information. Since Burst ORAM gives online IO strict priority over shuffle IO, our use of semaphores is substantially different than ObliviStore's, which tries to issue the same amount of IO after each request. ObliviStore uses four semaphores: *Shuffling Buffer*, *Early Cache-ins*, *Eviction*, and *Shuffling IO*. In Burst ORAM, we use three:

- *Shuffle Buffer* manages client space reserved for blocks from active shuffle jobs, and differs from ObliviStore's *Shuffling Buffer* only in initial value.
- *Local Space* manages all remaining space, combining ObliviStore's *Early Cache-in* and *Eviction* semaphores.
- *Concurrent IO* manages concurrent block transfers based on network link capacity, preventing the Shuffler from starving the Requester. It differs fundamentally from ObliviStore's *Shuffling IO* semaphore, which manages per-request shuffle IO.

***Shuffle Buffer* semaphore.** *Shuffle Buffer* gives the number of blocks that may be added to the client's shuffle buffer. We initialize it to double the maximum partition size (under $2.4\sqrt{N}$ total for $N > 2^{10}$), to ensure that the shuffle buffer is large enough to store at least two in-progress shuffle jobs. When *Shuffle Buffer* reaches 0, the Shuffler may not issue additional reads.

***Local Space* semaphore.** *Local Space* gives the number of blocks that may still be stored in remaining client space (space not reserved for the position map or shuffle buffer). If *Local Space* is 0, the Requester may not fetch more blocks. Blocks fetched by the Requester count toward *Local Space* until their partition's shuffle job is activated and they are absorbed into *Shuffle Buffer*. Once a block moves from *Local Space* to *Shuffle Buffer*, it is considered *free* from the client, and more requests may be issued. The more client space, the higher *Local Space*'s initial value, and the better our burst performance.

***Concurrent IO* semaphore.** *Concurrent IO* is initialized to the network link's block capacity. Queuing a block transfer decrements *Concurrent IO*, and completing a transfer increments *Concurrent IO*. The Shuffler may only initiate a transfer if *Concurrent IO* > 0. However, the Requester may continue to initiate transfers and decrement *Concurrent IO* even if it is negative. This mechanism ensures that no new shuffle IO starts while there is sufficient online IO to fully utilize the link. If no online IO starts, *Concurrent IO* eventually becomes positive, and shuffle IO resumes, ensuring full utilization.

6.3 Detailed System Behavior

We now describe the interaction between ORAM Main, the Requester, the Shuffler, and the semaphores in detail. Accompanying pseudocode can be found in Appendix A.

ORAM Main (Algorithm 1). Incoming read and write requests are asynchronously added to the Request Queue. During each iteration, ORAM Main first tries to advance the Requester, which attempts to satisfy the next request from the Request Queue. If the queue is empty, or *Local Space* too low, ORAM Main advances the Shuffler instead. This mechanism suppresses new shuffle IO during a new burst of requests until the Requester has fetched as many blocks as possible.

For each request, we evict v blocks to randomly chosen partitions, where v is the *eviction rate*, set to 1.3 as in ObliviStore [23]. When evicting, if the Requester has previously assigned a block to be evicted to partition p, then we evict that block. If there are no assigned blocks, then to maintain obliviousness we evict a new dummy block instead. Eviction does not send a block to the server immediately. It merely informs the Shuffler that

the block is ready to be shuffled into p.

Requester (Algorithm 2). To service a request, the Requester first identifies the partition and level containing the desired block. It then determines which levels require early shuffle reads, and which need only standard reads. If *Local Space* is large enough to accommodate the retrieved blocks, the requester issues an asynchronous request for the necessary blocks Else, control returns to ORAM Main, giving the Shuffler a chance to free space.

The server asynchronously returns the early shuffle read blocks and a single *combined* block obtained from all standard-read blocks using the XOR technique (Section 4). The Requester extracts the desired block from the combined block or from an early shuffle read block, then updates the block (write) or returns it to the client (read). The Requester then assigns the desired block for eviction to a randomly chosen partition.

Shuffler (Algorithm 3). The Shuffler may only proceed if *Concurrent IO* > 0. Otherwise, there is pending online IO, which takes priority over shuffle IO, so control returns to ORAM Main without any shuffling.

The Shuffler places shuffle jobs into three queues based on phase. The *New Job Queue* holds inactive jobs, prioritized by efficiency. The *Read Job Queue* holds active jobs for which some reads have been issued, but not all reads are complete. The *Write Job Queue* holds active jobs for which all reads, not writes, are complete.

If all reads have been issued for all jobs in the *Read Job Queue*, the Shuffler *activates* the most efficient job from the *New Job Queue*, if any. *Activating* a job moves it to the *Read Job Queue* and freezes its read/write levels, preventing it from being updated by subsequent evictions. It also moves the job's eviction and early shuffle read blocks from *Local Space* to *Shuffle Buffer*, freeing up *Local Space* to handle online requests. By ensuring that all reads for all active jobs are issued before activating new jobs, we avoid hastily activating inefficient jobs.

The Shuffler then tries to decrement *Shuffle Buffer* to determine whether a shuffle read may be issued. If so, the Shuffler asynchronously fetches a block for a job in the *Read Job Queue*. If not, the Shuffler asynchronously writes a block from a job in the *Write Job Queue* instead. Unlike reads, writes do not require *Shuffle Buffer* space, so they can always be issued. The Shuffler prioritizes reads since they are critical prerequisites to activating new jobs and freeing up *Local Space*. The equally costly writes can be delayed until *Shuffle Buffer* space runs out.

Once all reads for a job complete, the job is *shuffled*: dummy blocks are added as needed, then all are permuted and re-encrypted. We then move the job to the *Write Job Queue*. When all writes finish, we mark the job complete and remove it from the *Write Job Queue*.

6.4 Burst ORAM Security

We assume the server knows *public* information such as the values of each semaphore and the start and end times of each request. The server also knows the level configuration of each partition and the size and phase of each shuffle job, including which encrypted blocks have been read from and written to the server. We must prevent the server from learning the contents of any encrypted block, or anything about which plaintext block is being requested. Thus, the server may not know the location of a given plaintext block, or even the prior location of any previously requested encrypted block.

All of Burst ORAM's publicly visible actions are, or appear to the server to be, independent of the client's sensitive data access sequence. Since Burst ORAM treats the server as a simple block store, the publicly visible actions consist entirely of deciding when to transfer which blocks. Intuitively, we must show that each action taken by Burst ORAM is either deterministic and dependent only on public information, or appears random to the server. Equivalently, we must be able to generate a sequence of encrypted block transfers that appears indistinguishable from the actions of Burst ORAM using only public information. We now show how each Burst ORAM component meets these criteria.

ORAM Main & Client Security. ORAM Main (Algorithm 1) chooses whether to advance the Requester or the Shuffler, and depends on the size of the request queue and the *Local Space* semaphore. Since the number of pending requests and the semaphores are public, ORAM Main is deterministic and based only on public information. For each eviction, the choice of partition is made randomly, and exactly one block will always be evicted. Thus, every action in Algorithm 1 is either truly random or based on public information, and is trivial to simulate.

Requester Security. The Requester (Algorithm 2) must first identify the partition containing a desired block. Since the block was assigned to the partition randomly and this is the first time it is being retrieved since it was assigned, the choice of partition appears random to the server. Within each partition, the requester deterministically retrieves one block from each occupied level. The choice from each level appears random, since blocks were randomly permuted when the level was created.

The Requester singles out early shuffle reads and returns them individually. The identity of levels that return early shuffle reads is public, since it depends on the number of blocks in the level. The remaining blocks are deterministically combined using XOR into a single returned block. Finally, the request is marked satisfied only after all blocks have been returned, so request completion time depends only on public information.

The Requester's behavior can be simulated using only

public information by randomly choosing a partition and randomly selecting one block from each occupied level. Blocks from levels with at most half their original blocks remaining should be returned individually, and all others combined using XOR and returned. Once all blocks have been returned, the request is marked satisfied.

Shuffler Security. As in ObliviStore, Shuffler (Algorithm 3) operations depend on public semaphores. Job efficiency, which we use for prioritizing jobs, depends on the number of blocks to be read and written to perform shuffling, as well as the number of early shuffle reads and blocks already *evicted* (not *assigned*). The identity of early shuffle read levels and the number of evictions is public. Further, the number of reads and writes depends only on the partition's level configuration. Thus, job efficiency and job order depend only on public information. Since the Shuffler's actions are either truly random (e.g. permuting blocks) or depend only on public information (i.e. semaphores), it is trivial to simulate.

Client Space. Since fetched blocks are assigned randomly to partitions, but evicted using an independent process, the number of blocks awaiting eviction may grow. The precise number of such blocks may leak information about where blocks were assigned, so it must be kept secret, and the client must allocate a fixed amount of space dedicated to storing such blocks (see *Overflow Space* in Figure 6). ObliviStore [23] relies on a probabilistic bound on overflow space provided in [24]. Since Burst ORAM uses ObliviStore's assignment and eviction processes, the bound holds for Burst ORAM as well. Level caching uses space controlled by the *Local Space* semaphore, so it depends only on public information.

7 Evaluation

We ran simulations comparing response times and bandwidth costs of Burst ORAM with ObliviStore and an insecure baseline, using real and synthetic workloads.

7.1 Methodology

7.1.1 Baselines

We compare Burst ORAM and its variants against two baselines. The first is the ObliviStore ORAM described in [23], including its level compression optimization. For fairness, we allow ObliviStore to use extra client space to locally cache the smallest levels in each partition. The second baseline is an insecure scheme without ORAM in which blocks are encrypted, but access patterns are not hidden. It transfers exactly one block per request.

We evaluate Burst ORAM against ObliviStore since ObliviStore is the most bandwidth-efficient existing ORAM scheme. Other schemes require less client storage [25], but incur higher bandwidth costs, and thus would yield higher response times. We did not include results from Path-PIR [17] because it requires substantially larger block sizes to be efficient, and its response times are dominated by the orthogonal consideration of PIR computation. Path-PIR reports response times in the 40–50 second range for comparably-sized databases.

7.1.2 Metrics

We evaluate Burst ORAM and our baselines using *response time* and *bandwidth cost* as metrics (see Section 2). We measure average, maximum, and p-percentile response times for various p. A p-percentile response time of t seconds indicates that p percent of the requests were satisfied with response times under t seconds.

We explicitly measure online, effective, and overall bandwidth costs. In the insecure baseline, all are 1X, so response times are minimal. However, if a burst has high enough frequency to saturate available bandwidth, requests may still pile up, yielding large response times.

7.1.3 Workloads

We use three workloads. The first consists of an endless burst of requests all issued at once, and compares changes in bandwidth costs of each scheme as a function of burst length. The second consists of two identical bursts with equally-spaced requests, separated by an idle period. It shows how response times change in each scheme before and after the idle period.

The third workload is based on the NetApp Dataset [2, 14], a corporate workload containing file system accesses from over 5000 corporate clients and 2500 engineering clients during 100 days. The file system uses 22TB of its 31TB of available space. More details about the workload are provided in the work by Leung et al. [14].

Our NetApp workload uses a 15 day period (Sept. 25 through Oct. 9) during which corporate and engineering clients were active. Requested chunk sizes range from a few bits to 64KB, with most at least 4KB [14]. Thus, we chose a 4KB block size. In total, 312GB of data were requested using $8.8 \cdot 10^7$ 4KB queries.

We configure the NetApp workload ORAM with a 32TB capacity, and allow 100GB of client space, for a usable storage increase of 328 times. For Burst ORAM and ObliviStore, at least 33GB is consumed by the position map, and only 64GB is used for local block storage. The total block count is $N = 2^{33}$. Blocks are divided into $\lfloor 2^{17}/3 \rfloor$ partitions to maximize server space utilization, each with an upper-bound partition size of 2^{18} blocks.

7.2 Simulator

We evaluated Burst ORAM's bandwidth costs and response times using a detailed simulator written in Java. The simulator creates an asynchronous event for each block to be transferred. We calculate the transfer's expected end time from the network latency, the network bandwidth, and the number of pending transfers.

Our simulator also measures results for ObliviStore and the insecure baseline. In all schemes, block requests are time-stamped as soon as they arrive, and serviced as soon as possible. Requests pile up indefinitely if they arrive more frequently than the scheme can handle them.

Burst ORAM's behavior is driven by semaphores and appears data-independent to the server. Each request reads from a partition that appears to be chosen uniformly at random, so bandwidth costs and response times depend only on request arrival times, not on requested block IDs or contents. Thus, the simulator need only store counters representing the number of remaining blocks in each level of each partition, and can avoid storing block IDs and contents explicitly.

Since the simulator need not represent blocks individually, it does not measure the costs of encryption, looking up block IDs, or performing disk reads for blocks. Thus, measured bandwidth costs and response times depend entirely on network latency, bandwidth capacity, request arrival times, and the scheme itself.

7.2.1 Extrapolating Results to Real-World Settings

Burst ORAM can achieve near-optimal performance for realistic bursty traffic patterns. In particular, in many real-life cases bandwidth is overprovisioned to ensure near-optimal response time under bursts – for the insecure baseline. However, in between bursts, most of the bandwidth is not utilized. Burst ORAM's idea is leveraging the available bandwidth in between bursts to ensure near-optimal response time during bursts.

Our simulation applies mainly to scenarios where the client-server bandwidth is the primary bandwidth bottleneck (i.e., client-server bandwidth is the narrowest pipe in the system), which is likely to be the case in a real-life outsourced storage scenario, such as a corporate enterprise outsourcing its storage to a cloud provider. While the simulation assumes that there is a single server, in practice, the server-side architecture could be more complicated and involve multiple servers interacting with each other. But as long as server-server bandwidth is not the bottleneck, our simulation results would be applicable. Similarly, we assume that the server's disk bandwidth is not a bottleneck. This is likely the case if fast Solid State Drives (SSD) are employed. For example, assuming 4KB blocks and only one such array of SSDs with a 100μs random 4KB read latency, our single-array throughput limits us to satisfying $10,000$ requests per second. In contrast, even a 1Gbps network connection lets us satisfy only $32,000$ requests per second. Thus, with even six such arrays ($3\log_2 N$ SSDs total), assigning roughly $\sqrt{N}/6$ partitions to each array, we can expect the client-server network to be the bottleneck.

Other than bandwidth, another factor is inherent system latencies, e.g., network round-trip times, or inherent disk latencies. Under the same overall bandwidth configuration, increased latency is unlikely to affect the near-optimality of Burst ORAM– while they would increase Burst ORAM's total response times, we would expect a comparable increase in response times for the insecure baseline.

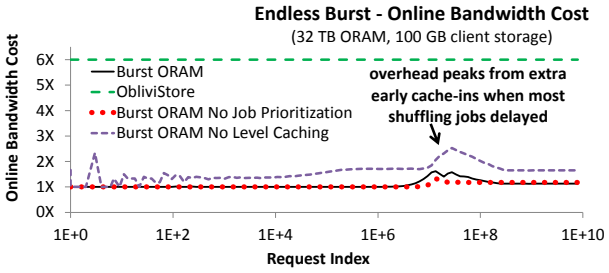


Figure 7: Online bandwidth costs as a burst lengthens. Burst ORAM maintains low online cost regardless of burst length, unlike ObliviStore.

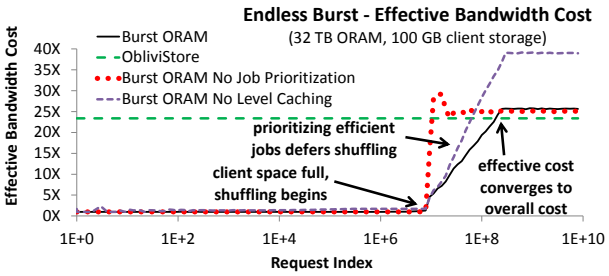


Figure 8: Effective bandwidth costs as burst grows. Burst ORAM handles most bursts with $\sim$1X effective cost. Effective costs converge to overall costs for long bursts.

7.3 Endless Burst Experiments

For the endless burst experiments, we use a 32TB ORAM with $N = 2^{33}$ 4KB blocks and 100GB client space. We issue 2^{33} requests at once, then start satisfying requests in order using each scheme. We record the bandwidth costs of each request, averaged over requests with similar indexes and over three trials. Figures 7 and 8 show online and effective costs, respectively. The insecure baseline is not shown, since its online, effective, and overall bandwidth costs are all 1.

Figure 7 shows that Burst ORAM maintains 5X–6X lower online cost than ObliviStore for bursts of all lengths. When Burst ORAM starts to delay shuffling, it incurs more early shuffle reads, increasing online cost, but stays well under 2X on average. Burst ORAM effective costs can be near 1X because writes associated with requests are not performed until blocks are shuffled.

Burst ORAM defers shuffling, so its effective cost stays close to its online cost until client space fills, while ObliviStore starts shuffling immediately, so its effective

cost stays constant (Figure 8). Thus, response times for short bursts will be substantially lower in Burst ORAM than in ObliviStore.

Eventually, client space fills completely, and even Burst ORAM must shuffle continuously to keep up with incoming requests. This behavior is seen at the far right of Figure 8, where each scheme's effective cost converges to its overall cost. Burst ORAM's XOR technique results in slightly higher overall cost than ObliviStore's level compression, so Burst ORAM is slightly less efficient for very long bursts. Without local level caching, Burst ORAM spends much more time shuffling the smallest levels, yielding the poor performance of *Burst ORAM No Level Caching*.

If shuffle jobs are started in arbitrary order, as for *Burst ORAM No Prioritization*, the amount of shuffling per request quickly increases, pushing effective cost toward overall cost. However, by prioritizing efficient shuffle jobs as in *Burst ORAM* proper, more shuffling can be deferred, keeping effective costs lower for longer, and maintaining shorter response times.

7.4 Two-Burst Experiments

Our Two-Burst experiments show how each scheme responds to idle time between bursts. We show that Burst ORAM uses the idle time effectively, freeing up as much client space as possible. The longer the gap between bursts, the longer Burst ORAM maintains low effective costs during Burst 2.

Figure 9 shows response times during two closely-spaced bursts, each of $\sim 2^{27}$ requests spread evenly over 72 seconds. The ORAM holds $N = 2^{25}$ blocks, and the client has space for 2^{18} blocks. Since we must also store early shuffle reads and reserve space for the shuffle buffer, the client space is not quite enough to accommodate a single burst entirely. We simulate a 100Mbps network connection with 50ms latency.

All ORAMs start with low response times during Burst 1. ObliviStore response times quickly increase due to fixed shuffle work between successive requests. Burst ORAMs delay shuffle work, so response times stay low until client space fills. Without level caching, additional early shuffle reads cause early shuffling and thus premature spikes in response times.

When Burst 1 ends, the ORAMs continue working, satisfying pending requests and catching up on shuffling during the idle period. Longer idle times allow more shuffling and lower response times at the start of Burst 2. None of the ORAMs have time to fully catch up, so response times increase sooner during Burst 2. ObliviStore cannot even satisfy all Burst 1 requests before Burst 2 starts, so response times start high on Burst 2. Burst ORAM does satisfy all Burst 1 requests, so it uses freed client space to efficiently handle early Burst 2 requests.

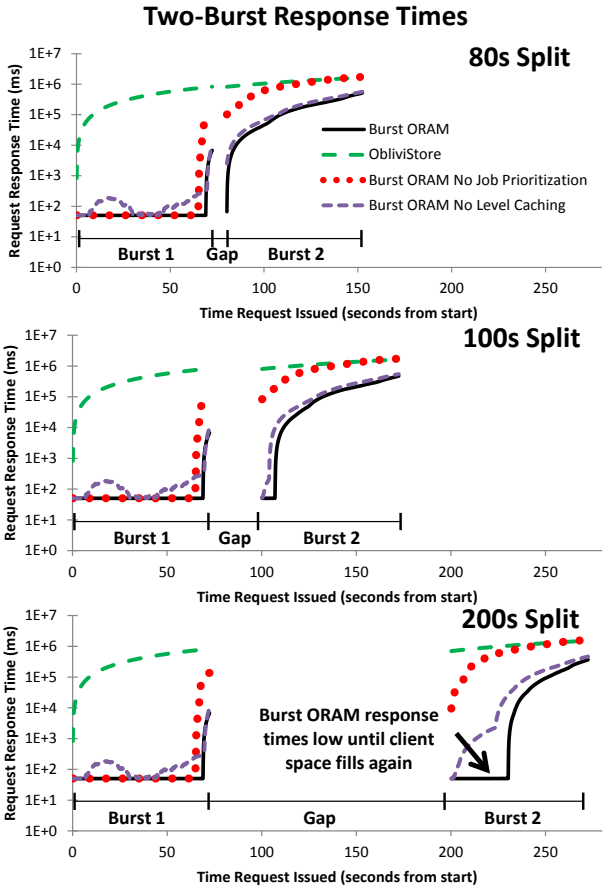


Figure 9: Response times during two same-size bursts of just over 2^{17} requests spread evenly over 72 seconds. Client has space for at most 2^{18} blocks. No level caching causes early spikes due to extra early shuffle reads.

Clearly, Burst ORAM performs better with shuffle prioritization, as it allows more shuffling to be delayed to the idle period, satisfying more requests quickly in both bursts. Burst ORAM also does better with local level caching. Without level caching, we start with more available client space, but the extra server levels yield more early shuffle reads to store, filling client space sooner.

7.5 NetApp Workload Experiments

The NetApp experiments show how each scheme performs on a realistic, bursty workload. Burst ORAM exploits the bursty request patterns, minimizing online IO and delaying shuffle IO to achieve near-optimal response times far lower than ObliviStore's. Level caching keeps Burst ORAM's overall bandwidth costs low.

Figure 10 shows 99.9-percentile response times for several schemes running the 15-day NetApp workload for varying bandwidths. All experiments assume a 50ms network latency. For most bandwidths, Burst ORAM response times are orders of magnitude lower than those

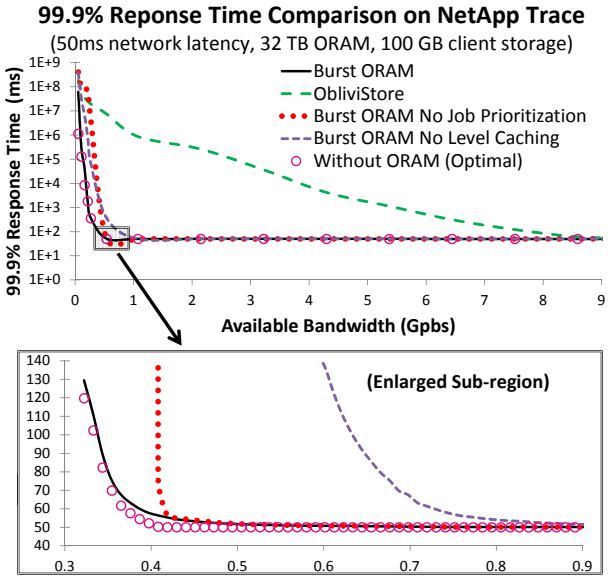


Figure 10: (Top) Burst ORAM achieves short response times in bandwidth-constrained settings. Since ObliviStore has high effective cost, it requires more available client-server bandwidth to achieve short response times. (Bottom) Burst ORAM response times are comparable to those of the insecure (without ORAM) scheme.

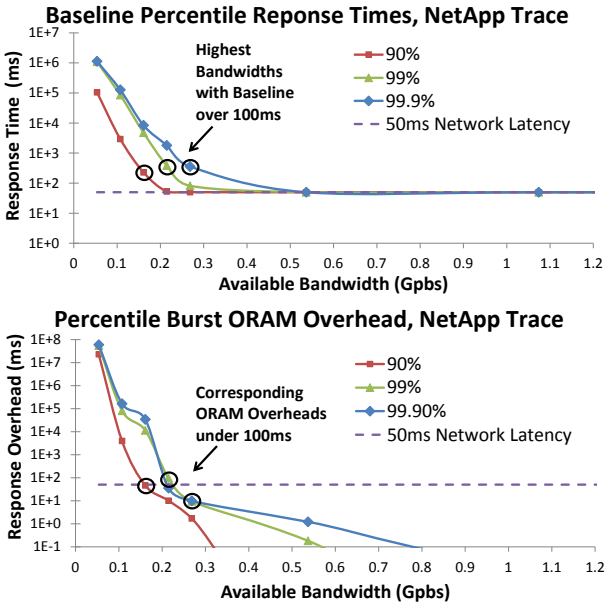


Figure 11: (Top) Insecure baseline (no ORAM) p-percentile response times for various p. (Bottom) Overhead (difference) between insecure baseline and Burst ORAM's p-percentile response times. Marked nodes show that when baseline p-percentile response times are $<$ 100ms, Burst ORAM overhead is also $<$ 100ms.

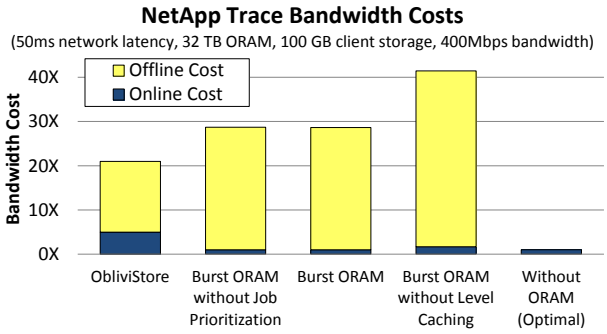


Figure 12: To achieve shorter response times, Burst ORAM incurs higher overall bandwidth cost than ObliviStore, most of which is consumed during idle periods. Level caching keeps bandwidth costs in check. Job prioritization does not affect overall cost, but does reduce effective costs and response times (Figures 8, 10).

of ObliviStore and comparable to those of the insecure baseline. Shuffle prioritization and level caching noticeably reduce response times for bandwidths under 1Gbps.

Figure 11 compares p-percentile response times for p values of 90%, 99%, and 99.9%. It gives absolute p-percentile response times for the insecure baseline, and differences between the insecure baseline and Burst ORAM p-percentile response times (Burst ORAM *overhead*). When baseline response times are low, Burst ORAM response times are also low across multiple p.

The NetApp dataset descriptions [2, 14] do not specify the total available network bandwidth, but since it was likely sufficient to allow decent performance, we expect from Figure 10 that it was at least between 200Mbps and 400Mbps. Figure 12 compares the overall bandwidth costs incurred by each scheme running the NetApp workload at 400Mbps. Costs for other bandwidths are similar. Burst ORAM clearly achieves an online cost several times lower than ObliviStore's.

Level caching reduces Burst ORAM's overall cost from 42X to 29X. Burst ORAM's higher cost is due to a combination of factors needed to achieve short response times. First, Burst ORAM uses the XOR technique, which is less efficient overall than ObliviStore's mutually exclusive level compression. Second, Burst ORAM handles smaller jobs first. Such jobs are more efficient in the short-term, but since they frequently write blocks to small levels, they create more future shuffle work. In ObliviStore, such jobs are often delayed during a large job, so fewer levels are created, reducing overall cost.

8 Conclusion

We have presented Burst ORAM, a novel Oblivious RAM scheme based on ObliviStore and tuned for practical response times on bursty workloads. We presented a

novel ORAM architecture for prioritizing online IO, and introduced the XOR technique for reducing online IO. We also introduced a novel scheduling mechanism for delaying shuffle IO, and described a level caching mechanism that uses extra client space to reduce overall IO.

We simulated Burst ORAM on a real-world workload and showed that it incurs low online and effective bandwidth costs during bursts. Burst ORAM achieved near-optimal response times that were orders of magnitude lower than existing ORAM schemes.

Acknowledgements.. This work was supported in part by grant N00014-07-C-0311 from ONR, the National Physical Science Consortium Graduate Fellowship; by NSF under grant number CNS-1314857, a Sloan Research Fellowship, a Google Faculty Research Award; by the NSF Graduate Research Fellowship under Grant No. DGE-0946797, a DoD National Defense Science and Engineering Graduate Fellowship, an Intel award through the ISTC for Secure Computing, and a grant from Amazon Web Services.

References

[1] BONEH, D., MAZIERES, D., AND POPA, R. A. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[2] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proc. ACM SOSP* (2011).

[3] CHOW, R., GOLLE, P., JAKOBSSON, M., SHI, E., STADDON, J., MASUOKA, R., AND MOLINA, J. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proc. ACM CCSW* (2009), pp. 85–90.

[4] DAMGÅRD, I., MELDGAARD, S., AND NIELSEN, J. B. Perfectly secure oblivious RAM without random oracles. In *TCC* (2011), pp. 144–163.

[5] DAUTRICH, J., AND RAVISHANKAR, C. Compromising privacy in precise query protocols. In *Proc. EDBT* (2013).

[6] FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proc. ACM CCS Workshop on Scalable Trusted Computing* (2012), pp. 3–8.

[7] GENTRY, C., GOLDMAN, K., HALEVI, S., JULTA, C., RAYKOVA, M., AND WICHS, D. Optimizing ORAM and using it efficiently for secure computation. In *PETS* (2013).

[8] GOLDREICH, O. Towards a theory of software protection and simulation by oblivious rams. In *STOC* (1987).

[9] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[10] GOODRICH, M., AND MITZENMACHER, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. *Automata, Languages and Programming* (2011), 576–587.

[11] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proc. SODA* (2012), SIAM, pp. 157–167.

[12] ISLAM, M., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proc. NDSS* (2012).

[13] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proc. SODA* (2012), SIAM, pp. 143–156.

[14] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *Proc. USENIX ATC* (2008), USENIX Association, pp. 213–226.

[15] LORCH, J. R., PARNO, B., MICKENS, J. W., RAYKOVA, M., AND SCHIFFMAN, J. Shroud: Ensuring private access to large-scale data in the data center. *FAST* (2013), 199–213.

[16] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical oblivious computation in a secure processor. In *ACM CCS* (2013).

[17] MAYBERRY, T., BLASS, E.-O., AND CHAN, A. H. Efficient private file retrieval by combining ORAM and PIR. In *NDSS* (2014).

[18] OSTROVSKY, R., AND SHOUP, V. Private information storage (extended abstract). In *STOC* (1997), pp. 294–303.

[19] PINKAS, B., AND REINMAN, T. Oblivious RAM revisited. In *CRYPTO* (2010).

[20] REN, L., YU, X., FLETCHER, C. W., VAN DIJK, M., AND DEVADAS, S. Design space exploration and optimization of path oblivious RAM in secure processors. In *Proc. ISCA*. 2013.

[21] SHI, E., CHAN, H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT* (2011).

[22] STEFANOV, E., AND SHI, E. Multi-Cloud Oblivious Storage. In *CCS* (2013).

[23] STEFANOV, E., AND SHI, E. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy* (2013).

[24] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious RAM. NDSS, 2012.

[25] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An extremely simple oblivious RAM protocol. In *ACM CCS* (2013).

[26] WILLIAMS, P., AND SION, R. Sr-oram: Single round-trip oblivious ram. *ACNS, industrial track* (2012), 19–33.

[27] WILLIAMS, P., SION, R., AND CARBUNAR, B. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. ACM CCS* (2008), pp. 139–148.

[28] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: A parallel oblivious file system. In *CCS* (2012).

A Pseudocode

Algorithms 1–4 give pseudocode for Burst ORAM, using the notation summarized in Table 1. The algorithms are described in detail in Section 6, but we clarify some of the code and notation below.

The efficiency of shuffle job J_p is given by:

$$E_{J_p} = \frac{V_{J_p} + A_{J_p}}{R_{J_p} + W_{J_p}}. \quad (1)$$

C_p represents the state of partition p at the time p's last shuffle job completed, and determines the current set of occupied levels in p. V_p represents the number of blocks that have been evicted to p, since p's last shuffle job completed. $C_p + V_p$ determines which levels would be occupied if p were to be completely shuffled.

V_{J_p} represents the number of evicted blocks that will be shuffled into p by J_p. Thus, C_p and V_{J_p} together determine J_p's read and write levels.

If J_p is inactive, it is updated whenever V_p changes, setting $V_{J_p} \leftarrow V_p$ (Algorithm 1, Line 25). However, we

Table 1: Algorithm Notation

v	Eviction rate: blocks evicted per request
λ	Number of levels cached locally
p	A partition
V_p	# blocks evicted to p since p's last shuffle end
C_p	p's state after last shuffle (shuffled evictions)
b	Block ID
$D(b)$	Plaintext contents of b
$E(b)$	Encrypted contents of b
$S(b)$	Server address/ID of b
$P(b)$	Partition containing b, or random if none
$L(b)$	Level containing b, or $\perp$ if none
Q	IDs of standard-read blocks to fetch
C	IDs of early shuffle read blocks to fetch
X_Q	Combined block (XOR of all blocks in Q)
X'_Q	Subtraction block (XOR of dummies in Q)
J_p	Shuffle job for p
V_{J_p}	Number of evicted blocks J_p will shuffle
E_{J_p}	Efficiency of J_p
A_{J_p}	Number of early shuffle reads for J_p
R_{J_p}	Total blocks remaining to be read for J_p
W_{J_p}	Total blocks to write for J_p
NJQ	New Job Queue
RJQ	Read Job Queue
WJQ	Write Job Queue

implement level caching by skipping those updates to J_p that would cause J_p to write to levels with indexes less than λ (Algorithm 1, Line 23). Once J_p is active, V_{J_p} is no longer updated. When J_p completes, p's state is updated to reflect the blocks shuffled in by J_p, setting $C_p \leftarrow C_p + V_{J_p}$ (Algorithm 3, Line 37).

If p has no inactive shuffle job, the job is created following the first eviction to p that would allow updating (Algorithm 1, Line 24). If p has no active job, the inactive job moves to the *New Job Queue* (NJQ) as soon as the job is created (Algorithm 1, Line 27), where it stays until the job is activated. If p does have an active shuffle job, the inactive job is not added to NJQ until the active job completes (Algorithm 3, Line 38).

Thus, NJQ contains only inactive shuffle jobs for those partitions with no active job, ensuring that any job in NJQ may be activated. NJQ is a priority queue serving the most efficient jobs first. Job efficiency may change while the job is in NJQ, since V_{J_p} can still be updated.

B Reducing Online Costs of SR-ORAM

We now briefly describe how SR-ORAM [26] can benefit from our XOR technique. Like ObliviStore, SR-ORAM requires only a single round-trip to satisfy a request, and has online bandwidth cost $O(\log N)$. SR-ORAM uses an

Algorithm 1 Pseudocode for Client and ORAM Main

```
1:  function ClientRead(b)
2:      Append b to RequestQueue
3:      On RequestCallback(D(b)), return D(b)
4:  procedure Write(b, d)
5:      Append b to RequestQueue
6:      On RequestCallback(D(b)), write d to D(b)
7:  procedure ORAM Main
8:      RequestMade ← false
9:      if RequestQueue ≠ ∅ then
10:         b ← Peek(RequestQueue)
11:         if Fetch(b) then                ▷ Request Issued
12:             RequestMade ← true
13:             Pop(RequestQueue)
14:             MakeEvictions
15:     if RequestMade = false then
16:         TryShuffleWork
17: procedure MakeEvictions
18:     PendingEvictions = PendingEvictions + v
19:     while PendingEvictions ≥ 1 do
20:         p ← random partition
21:         Evict new dummy or assigned real block to p
22:         V_p = V_p + 1
23:         if shuffling p only writes levels ≥ λ then
24:             J_p ← p's inactive job ▷ Create if needed
25:             V_{J_p} ← V_p
26:             if p has no active job then
27:                 NJQ = NJQ ∪ J_p
28:         PendingEvictions = PendingEvictions − 1
```

encrypted Bloom filter to let the server obliviously check whether each level contains the requested block. The server retrieves the requested block from its level, and client-selected dummies from all others. Since at most one block is real, the server can XOR all the blocks together and return a single combined block.

One difference in SR-ORAM is that the client *does not* know a priori which level contains the requested block. Thus, SR-ORAM must be modified to include the level index of each retrieved block in its response. To allow the client to easily reconstruct dummies, we must also change SR-ORAM to generate the contents of each dummy block as in Burst ORAM. Since the client knows the indexes of the dummy blocks it requested from each level, it can infer the real block's level from the server's response. The client then reconstructs the all dummy block contents and XORs them with the returned block to obtain the requested block, as in Burst ORAM.

SR-ORAM is a synchronous protocol, so it has no notion equivalent to early shuffle reads. Thus, the XOR technique reduces SR-ORAM's online bandwidth cost from $O(\log N)$ to 1. The reduction in overall

Algorithm 2 Pseudocode for Requester

```
1:  function FETCH(b)
2:      P(b), L(b) ← position map lookup on b
3:      Q = ∅, C = ∅
4:      for level ℓ ∈ P(b) do
5:          if ℓ is non-empty then
6:              b_ℓ ← b if ℓ = L(b)
7:              b_ℓ ← ID of next dummy in ℓ if ℓ ≠ L(b)
8:              if ℓ more than half full then
9:                  Q ← Q ∪ S(b_ℓ)            ▷ Standard read
10:             else
11:                 C ← C ∪ S(b_ℓ)            ▷ Early shuffle read
12:     Ret ← |C| + MAX(|Q|, 1)               ▷ # blocks to return
13:     if Not TRYDEC(Local Space, Ret) then
14:         return false                      ▷ Not enough space for blocks
15:     DEC(Concurrent IO, Ret)
16:     Issue asynch. request for (C, Q) to server
17:     When done, server calls:
18:         FETCHCALLBACK(E(C), XOR of E(Q))
19:     return true
20: procedure FETCHCALLBACK({E(c_i)}, X_Q)
21:     INC(Concurrent IO, 1)
22:     if b ∈ Q then
23:         X'_Q ← ⊕{E(q_i) | S(q_i) ∈ Q, q_i ≠ b}
24:                ▷ Subtraction block, computed locally
25:         E(b) ← X_Q ⊕ X'_Q
26:     if b ∈ C then
27:         E(b) ← E(c_i) where c_i = b
28:     D(b) ← decrypt E(b)
29:     Assign b for eviction to random partition
30:     REQUESTCALLBACK(D(b))
```

cost is negligible, as SR-ORAM has an offline cost $O(\log^2 N \log\log N)$. SR-ORAM contains only one hierarchy of $O(\log N)$ levels, so XOR incurs only $O(\log N)$ extra storage cost for the level-specific keys, fitting into SR-ORAM's logarithmic client storage.

Algorithm 3 Pseudocode for Shuffler

```
1:  procedure TRYSHUFFLEWORK
2:      if Not TRYDEC(Concurrent IO, 1) then
3:          return
4:      ReadIssued, WriteIssued ← false
5:      if All reads for jobs in RJQ issued then
6:          TRYACTIVATE                       ▷ Try to add job to RJQ
7:      if J_p ∈ RJQ has not issued read b_R then
8:          if TRYDEC(Shuffle Buffer, 1) then
9:              Issue asynch. request for S(b_R)
10:             When done: READCALLBACK(E(b_R))
11:             ReadIssued ← true
12:     if !ReadIssued and J_p ∈ WJQ has write b_W then
13:         Write E(b_W) to server
14:         When done, call WRITECALLBACK(S(b_W))
15:         WriteIssued ← true
16:     if Not ReadIssued and Not WriteIssued then
17:         INC(Concurrent IO, 1)             ▷ No shuffle work
18: procedure TRYACTIVATE
19:     if NJQ ≠ ∅ then
20:         J_p ← PEEK(NJQ)                   ▷ Most efficient job
21:         if TRYDEC(Shuffle Buffer, V_{J_p} + A_{J_p}) then
22:             Mark J_p active               ▷ V_{J_p} frozen
23:             INC(Local Space, V_{J_p} + A_{J_p})
24:             Move J_p from NJQ to RJQ
25: procedure READCALLBACK(E(b_R))
26:     INC(Concurrent IO, 1)
27:     Decrypt E(b_R), place D(b_R) in Shuffle Buffer
28:     if all reads in J_p have finished then
29:         Create dummy blocks to get W_{J_p} blocks total
30:         Permute and re-encrypt the blocks
31:         Move J_p from RJQ to WJQ
32: procedure WRITECALLBACK(S(b_W))
33:     INC(Concurrent IO, 1)
34:     if all writes in J_p have finished then
35:         Mark J_p complete
36:         Remove J_p from WJQ
37:         Update C_p ← C_p + V_{J_p}, V_p ← V_p − V_{J_p}
38:         Add p's inactive job, if any, to NJQ
```

Algorithm 4 Pseudocode for semaphores

```
1: procedure DEC(Semaphore, Quantity)
2:     Semaphore ← Semaphore − Quantity
3: procedure INC(Semaphore, Quantity)
4:     Semaphore ← Semaphore + Quantity
5: function TRYDEC(Semaphore, Quantity)
6:     if Semaphore < Quantity then return false
7:     DEC(Semaphore, Quantity); return true
```

TRUESET: Faster Verifiable Set Computations*

Ahmed E. Kosba[†§] akosba@cs.umd.edu
Dimitrios Papadopoulos[¶] dipapado@bu.edu
Charalampos Papamanthou[‡§] cpap@umd.edu
Mahmoud F. Sayed[†§] mfayoub@cs.umd.edu
Elaine Shi[†§] elaine@cs.umd.edu
Nikos Triandopoulos[‖¶] nikolaos.triandopoulos@rsa.com

Abstract

Verifiable computation (VC) enables thin clients to efficiently verify the computational results produced by a powerful server. Although VC was initially considered to be mainly of theoretical interest, over the last two years impressive progress has been made on implementing VC. Specifically, we now have open-source implementations of VC systems that handle *all* classes of computations expressed either as circuits or in the RAM model. Despite this very encouraging progress, new enhancements in the design and implementation of VC protocols are required to achieve truly practical VC for real-world applications.

In this work, we show that for functions that can be expressed efficiently in terms of set operations (e.g., a subset of SQL queries) VC can be enhanced to become drastically more practical: We present the design and prototype implementation of a novel VC scheme that achieves orders of magnitude speed-up in comparison with the state of the art. Specifically, we build and evaluate TRUESET, a system that can verifiably compute any polynomial-time function expressed as a circuit consisting of "set gates" such as *union*, *intersection*, *difference* and *set cardinality*. Moreover, TRUESET supports hybrid circuits consisting of both set gates and traditional arithmetic gates. Therefore, it does not lose any of the expressiveness of previous schemes—this also allows the user to choose the most efficient way to represent different parts of a computation. By expressing set computations as polynomial operations and introducing a novel Quadratic Polynomial Program technique, our experiments show that TRUESET achieves prover performance speed-up ranging from **30x** to **150x** and up to **97**% evaluation key size reduction compared to the state-of-the-art.

*This research was funded in part by NSF under grant numbers CNS-1314857, CNS-1012798 and CNS-1012910 and by a Google Faculty Research Award. The views and conclusions contained herein are those of the authors and do not represent funding agencies.

[†]Computer Science Dept., University of Maryland.
[‡]Electrical & Computer Engineering Dept., University of Maryland.
[§]U. Maryland Institute for Advanced Computer Studies (UMIACS).
[¶]Computer Science Dept., Boston University.
[‖]RSA Laboratories.

1 Introduction

Verifiable Computation (VC) is a cryptographic protocol that allows a client to outsource expensive computation tasks to a worker (e.g., a cloud server), such that the client can verify the result of the computation in less time than that required to perform the computation itself. Cryptographic approaches for VC [5, 6, 7, 12, 13, 14, 21] are attractive in that they require no special trusted hardware or software on the server, and can ensure security against arbitrarily malicious server behavior, including software/hardware bugs, misconfigurations, malicious insiders, and physical attacks.

Due to its various applications such as secure cloud computing, the research community has recently made impressive progress on Verifiable Computation, both on the theoretical and practical fronts. In particular, several recent works [2, 3, 9, 23, 25, 26, 29] have implemented Verifiable Computation for general computation tasks, and demonstrated promising evidence of its efficiency. Despite this encouraging progress, performance improvement of orders of magnitude is still required (especially on the time that the server takes to compute the proof) for cryptographic VC to become truly practical.

Existing systems for Verifiable Computation are built to accommodate any language in NP: Specifically, functions/programs are represented as either circuits (Boolean or arithmetic) or sets of constraints and cryptographic operations are run on these representations. While such an approach allows us to express any polynomial-time computation, it is often not the most efficient way to represent common computation tasks encountered in practice. For example, Parno et al. [23] point out that the behavior of their construction deteriorates abruptly for functionalities that have "bad" arithmetic circuit representation and Braun et al. [9] recognize that their scheme is not quite ready for practical use, restricting their evaluations to "smaller scales than would occur in real applications."

In order to reduce the practical cost of Verifiable Computation, we design and build TRUESET. TRUESET is an efficient and *provably secure* VC system that specializes in handling *set-centric* computation tasks. It allows us to

model computation as a *set circuit*—a circuit consisting of a combination of set operators (such as intersection, union, difference and sum), instead of just arithmetic operations (such as addition and multiplication in a finite field). For computation tasks that can be naturally expressed in terms of set operations (e.g., a subset of SQL database queries), our experimental results suggest *orders-of-magnitude* performance improvement in comparison with existing VC systems such as Pinocchio [23]. We now present TRUESET's main contributions:

Expressiveness. TRUESET retains the expressiveness of existing VC systems, in that it can support arbitrary computation tasks. Fundamentally, since our set circuit can support intersection, union, and set difference gates, the set of logic is complete[1]. Additionally, in Section 4.4, we show that TRUESET can be extended to support circuits that have a mixture of arithmetic gates and set gates. We achieve this by introducing a "split gate" (which, on input a set, outputs the individual elements) and a "merge gate" (which has the opposite function of the split gate).

Input-specific running time. One important reason why TRUESET significantly outperforms existing VC systems in practice is that it achieves *input-specific* running time for proof computation and key generation. Input-specific running time means that the running time of the prover is proportional to the size of the current input.

Achieving input-specific running time is not possible when set operations are expressed in terms of Boolean or arithmetic circuits, where one must account for worst-case set sizes when building the circuit: For example, in the case of intersection, the worst case size of the output is the minimum size of the two sets; in the case of union, the worst case size of the output is the sum of their sizes. Note that this applies not only to the set that comprises the final outcome of the computation, but to every intermediate set generated during the computation. As a result, existing approaches based on Boolean or arithmetic circuits incur a large blowup in terms of circuit size when used to express set operations. In this sense, TRUESET also achieves *asymptotic* performance gains for set-centric computation workloads in comparison with previous approaches.

TRUESET achieves input-specific running time by encoding a set of cardinality c as a polynomial of degree c (such an encoding was also used in previous works, e.g., [18, 22]), and a set circuit as a circuit on polynomials, where every wire is a polynomial, and every gate performs polynomial addition or multiplication. As a result, per-gate computation time for the prover (including the time for performing the computation and the time for producing the proof) is (quasi-)linear in the degree of the polynomial (i.e., cardinality of the actual set), and not proportional to the worst-case degree of the polynomial.

Finally, as in other VC systems, verifying in TRUESET requires work proportional to the size of inputs/outputs, but not in the running time of the computation.

[1] Any function computable by Boolean circuits can be computed by a set circuit: If one encodes the empty set as 0 and a fixed singleton set $\{s\}$ as 1, a union expresses the OR gate, an intersection expresses the AND gate and a set difference from $\{s\}$ expresses the NOT gate.

```
SELECT COUNT(UNIVERSITY.id)
    FROM UNIVERSITY JOIN CS
      ON UNIVERSITY.id = CS.id
```

Figure 1: An example of a JOIN *SQL query (between tables* **UNIVERSITY** *and* **CS***) that can be efficiently supported by* TRUESET. TRUESET *will implement* JOIN *with an intersection gate and* COUNT *with a cardinality gate.*

Implementation and evaluation. We implemented TRUESET and documented its efficiency comparing it with a verifiable protocol that compiles a set circuit into an arithmetic circuit and then uses Pinocchio [23] on the produced circuit. In TRUESET the prover's running time is reduced by approximately **30x** for all set sizes of 64 elements or more. In particular, for a single intersection/union gate over 2 sets of 256 elements each, TRUESET improves the prover cost by nearly **150x**. We also show that, while other systems [23] cannot—in a reasonable amount of time—execute over larger inputs, TRUESET can scale to large sets, e.g., sets with cardinality of approximately 8000 (2^{13}), efficiently accommodating instances that are about **30x** larger than previous systems. Finally, TRUESET greatly reduces the evaluation key size, a reduction that can reach **97**% for some operations.

Applications. TRUESET is developed to serve various information retrieval applications that use set operations as a building block. For example, consider an SQL query that performs a JOIN over two tables and then computes MAX or SUM over the result of the join operation. TRUESET can model the join operation as an intersection and then use the split gate to perform the maximum or the summation/cardinality operation over the output of the join—see Figure 1. Other queries that TRUESET could model are advanced keyword search queries containing complicated filters that can be expressed as arbitrary combinations of set operations (union, intersection, difference) over an underlying data set. Finally, the computation of similarity measurements for datasets often employs set operations. One of the most popular measurements of this type, is the Jaccard index [17] which is computed for two sets, as the ratio of the *cardinalities* of their intersection and union, a computation that can be easily compiled with TRUESET.

Technical highlight. Our core technical construction is inspired by the recent *quadratic span and arithmetic programs* [14], which were used to implement VC for any

Boolean or arithmetic circuit. Since our internal representation is a polynomial circuit (as mentioned earlier), we invent *quadratic polynomial programs* (QPP). During the prover's computation, polynomials on the wires of the circuit are evaluated at a random point s—however, this takes place in the exponent of a bilinear group, in a way that the server does not learn s. Evaluating the polynomial at the point s in effect reduces the polynomial to a value—therefore one can now think of the polynomial circuit as a normal arithmetic circuit whose wires encode plain values. In this way, we can apply techniques resembling quadratic arithmetic programs. While the intuition may be summarized as above, designing the actual algebraic construction and formally proving its security is nonetheless challenging, and requires a non-trivial transformation of quadratic arithmetic programs.

1.1 Related Work

There exists a large amount of theoretical work on VC: Micali [21] presented a scheme that can accommodate proofs for any language in NP. A more efficient approach is based on *succinct non-interactive arguments of knowledge* (SNARKs) [5, 6, 7, 14]. For the case of polynomial-time computable functions, protocols based on fully-homomorphic encryption [12, 13] and attribute-based encryption [24] have also been proposed. In general, the above schemes employ heavy cryptographic primitives and therefore are not very practical.

Recent works [2, 3, 9, 23, 25, 26, 29] have made impressive progress toward implementations of some of the above schemes, showing practicality for particular functionalities. Unfortunately, the server's cost for proof computation remains too high to be considered for wide deployment in real-world applications.

The problem of verifying a circuit of set operations was first addressed in a recent work by Canetti et al. [10]. Their proofs are of size linear to the size of the circuit, without however requiring a preprocessing phase for each circuit. In comparison, our proofs are of constant size, once such a preprocessing step has been run.

Papamanthou et al. [22] presented a scheme that provides verifiability for a single set operation. However, more general set operations can be accommodated by sequentially using their approach, since all intermediate set outputs are necessary for verification. This would lead to increased communication complexity.

A related scheme appears in the work of Chung et al. [11]. As this scheme uses Turing machines for the underlying computation model, the prover has inherently high complexity. Another work that combines verifiable computation with outsourcing of storage is [1], where a protocol for streaming datasets is proposed but the supported functionalities are quadratic polynomials only.

2 Definitions

In this section we provide necessary definitions and terminology that will be useful in the rest of the paper.

Circuits of sets and polynomials. TRUESET uses the same computation abstraction as the one used in the VC scheme by Parno et al. [23]: a circuit. However, instead of field elements, the circuit wires now carry *sets*, and, instead of arithmetic multiplication and addition gates, our circuit has three types of gates: *intersection*, *union* and *difference*. For the sake of presentation, the sets we are considering are simple sets, though our construction can be extended to support multisets as well. We therefore begin by defining a set circuit:

Definition 1 (Set circuit $\mathcal{C}$) *A set circuit $\mathcal{C}$ is a circuit that has gates that implement set union, set intersection or set difference over sets that have elements in a field $\mathbb{F}$.*

A set circuit is a tool that provides a clean abstraction of the computational steps necessary to perform a set operation. This structured representation will allow us to naturally encode a set operation into a number of execution conditions that are met when it is performed correctly. We stress that it is merely a theoretical abstraction and does not affect the way in which the computation is performed; the computing party can use its choice of efficient native libraries and architectures. In comparison, previous works that use arithmetic circuits to encode more general computations, require the construction (or simulation) and evaluation of such a circuit, an approach that introduces an additional source of overhead.

As mentioned in the introduction, our main technique is based on mapping any set circuit $\mathcal{C}$ to a circuit $\mathcal{F}$ of polynomial operations, i.e., to a circuit that carries univariate polynomials on its wires and has polynomial multiplication and polynomial addition gates. We now define the polynomial circuit $\mathcal{F}$:

Definition 2 (Polynomial circuit $\mathcal{F}$) *A polynomial circuit $\mathcal{F}$ in a field $\mathbb{F}$ is a circuit that has gates that implement univariate polynomial addition and univariate polynomial multiplication over $\mathbb{F}$. We denote with d the number of multiplication gates of $\mathcal{F}$ and with N the number of input and output wires of $\mathcal{F}$. The input and output wires are indexed $1, \ldots, N$. The rest of the wires[2] are indexed $N+1, \ldots, m$.*

SNARKs. TRUESET's main building block is a primitive called *succinct non-interactive argument of knowledge* (SNARK) [14]. A SNARK allows a client to commit to

[2] These wires include free wires (which are inputs only to multiplication gates) and the outputs of the internal multiplication gates (whose outputs are not outputs of the circuit). The set of these wires is denoted with I_m and has size at most $3d$.

a computation circuit C and then have a prover provide succinct cryptographic proofs that there exists an assignment on the wires w (which is called witness) such that the input-output pair $x = (\mathcal{I}, \mathcal{O})$ is valid.

As opposed to *verifiable computation* [24], a SNARK allows a prover to specify some wires of the input $\mathcal{I}$ as part of the witness w (this is useful when proving membership in an NP language, where the prover must prove witness existence). For this reason, SNARKs are more powerful than VC and therefore for the rest of the paper, we will show how to construct a SNARK for hierarchical set operations. In the full version of our paper [20], we show how to use the SNARK construction to provide a VC scheme as well as a VC scheme for outsourced sets, where the server not only performs the computation, but also stores the sets for the client. We now give the SNARK definition, adjusted from [14].

Definition 3 (SNARK scheme) *A* SNARK *scheme consists of three probabilistic polynomial time (PPT) algorithms* (KeyGen, Prove, Verify) *defined as follows.*

1. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^k, C)$. *The key generation algorithm takes as input the security parameter k and a computation circuit C; it outputs a public key* pk*, and a secret key* sk*.*

2. $\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$*: The prover algorithm takes as input the public key* pk*, an input-output pair $x = (\mathcal{I}, \mathcal{O})$, a valid witness w and it outputs a proof π.*

3. $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{sk}, x, \pi)$*: Given the key* sk*, a statement x and a proof π, the verification algorithm outputs 0 or 1.*

We say that a SNARK is publicly-verifiable if $\mathsf{sk} = \mathsf{pk}$*. In this case, proofs can be verified by anyone with* pk*. Otherwise, we call it a secretly-verifiable SNARK, in which case only the party with* sk *can verify.*

There are various properties that a SNARK should satisfy. The most important one is *soundness*. Namely, no PPT adversary should be able to output a verifying proof π for an input-output pair $x = (\mathcal{I}, \mathcal{O})$ that is not consistent with C. All the other properties of SNARKs are described formally in Appendix 6.2.

3 A SNARK for Polynomial Circuits

In their recent seminal work, Gennaro et al. [14] showed how to compactly encode computations as quadratic programs, in order to derive very efficient SNARKs. Specifically, they show how to convert any arithmetic circuit into a comparably-sized Quadratic Arithmetic Program (QAP), and any Boolean circuit into a comparably-sized Quadratic Span Program (QSP).

In this section we describe our SNARK construction for polynomial circuits. The construction is a modification of the optimized construction for arithmetic circuits that was presented by Parno et al. [23] (Protocol 2) and which is based on the original work of Gennaro et al. [14]. Our extension accounts for univariate polynomials on the wires, instead of just arithmetic values. We therefore need to define a *quadratic polynomial program*:

Definition 4 (Quadratic Polynomial Program (QPP)) *A QPP $\mathcal{Q}$ for a polynomial circuit $\mathcal{F}$ contains three sets of polynomials $\mathcal{V} = \{v_k(x)\}, \mathcal{W} = \{w_k(x)\}, \mathcal{Y} = \{y_k(x)\}$ for $k = 1, \ldots, m$ and a target polynomial $\tau(x)$. We say that $\mathcal{Q}$ computes $\mathcal{F}$ if: $c_1(z), c_2(z), \ldots, c_N(z)$ is a valid assignment of $\mathcal{F}$'s inputs and outputs iff there exist polynomials $c_{N+1}(z), \ldots, c_m(z)$ such that $\tau(x)$ divides $p(x, z)$ where*

$$p(x,z) = \left(\sum_{k=1}^{m} c_k(z) v_k(x)\right)\left(\sum_{k=1}^{m} c_k(z) w_k(x)\right) - \left(\sum_{k=1}^{m} c_k(z) y_k(x)\right). \quad (3.1)$$

We define the degree of $\mathcal{Q}$ to equal the degree of $\tau(x)$.

The main difference of the above quadratic program with the one presented in [23] is the fact that we introduce another variable z in the polynomial $p(x, z)$ representing the program (hence we need to account for bivariate polynomials, instead of univariate), which is going to account for the polynomials on the wires of the circuit.

Constructing a QPP. We now show how to construct a QPP $\mathcal{Q}$ for a polynomial circuit. The polynomials in $\mathcal{V}, \mathcal{W}, \mathcal{Y}$ and the polynomial $\tau(x)$ are computed as follows. Let $r_1, r_2, \ldots, r_d$ be random elements in $\mathbb{F}$. First, set $\tau(x) = (x - r_1)(x - r_2)\ldots(x - r_d)$ and compute the polynomial $v_k(x)$ such that $v_k(r_i) = 1$ iff wire k is the *left input* of multiplication gate i, otherwise $v_k(r_i) = 0$. Similarly, $w_k(r_i) = 1$ iff wire k is the *right* input of multiplication gate i, otherwise $w_k(r_i) = 0$ and $y_k(r_i) = 1$ iff wire k is the *output* of multiplication gate i, otherwise $y_k(r_i) = 0$. For example, consider the circuit of Figure 2 that has five inputs and one output and its wires are numbered as shown in the figure (gates take the index of the their output wire). Then $\tau(x) = (x - r_6)(x - r_7)$. For v_k we require that $v_k(r_6) = 0$ except for $v_2(r_6) = 1$, since the second wire is the only left input for the sixth gate, and $v_k(r_7) = 0$ except for $v_1(r_7)$ and $v_6(r_7)$ which are 1, since the first and sixth wire contribute as left inputs to gate 7. Right input polynomials w_k are computed similarly and output polynomials y_k are computed such that $y_6(r_6) = y_7(r_7) = 1$; all other cases are set to 0.

To see why the above QPP computes $\mathcal{F}$, let us focus on a single multiplication gate g, with k_1 being its

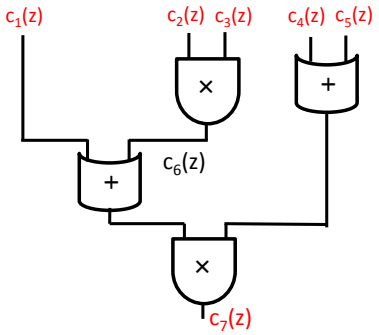


Figure 2: A sample polynomial circuit.

output wire and k_2 and k_3 be its left and right input wires respectively. Due to the divisibility requirement, it holds $p(r_i, z) = 0$ for $i = 1, \ldots, d$, hence Equation 3.1 will give $(\sum_{k=1}^{m} c_k(z)v_k(r_g))(\sum_{k=1}^{m} c_k(z)w_k(r_g)) = (\sum_{k=1}^{m} c_k(z)y_k(r_g))$. Now, from the way the polynomials v_k, w_k, y_k were defined above, most terms are 0 and what remains is $c_{k_2}(z)v_{k_2}(r_g) \cdot c_{k_3}(z)w_{k_3}(r_g) = c_{k_1}(z)y_{k_1}(r_g)$ or else $c_{k_2}(z) \cdot c_{k_3}(z) = c_{k_1}(z)$, which is the definition of a multiplication gate. More formally:

Lemma 1 *The above QPP $\mathcal{Q}$ computes $\mathcal{F}$.*

Proof: ($\Rightarrow$) Suppose $c_1(z), c_2(z), \ldots, c_N(z)$ are correct assignments of the input and output wires but there do not exist polynomials $c_{N+1}(z), \ldots, c_m(z)$ such that $\tau(x)$ divides $p(x, z)$. Then there is at least one multiplication gate r with left input x, right input y and output o, such that $p(r, z) \neq 0$. Let p be the path of multiplication gates that contains r starting from an input polynomial $c_i(z)$ to an output polynomial $c_j(z)$, where $i, j \leq N$. Since $c_i(z)$ and $c_j(z)$ are correct assignments, there must exist polynomials $c_x(z)$ and $c_y(z)$ such that $c_x(z)c_y(z) = c_o(z)$. Since r has a single left input, a single right input and a single output it holds $v_x(r) = 1$ and $v_i(r) = 0$ for all $i \neq x$. Similarly, $w_y(r) = 1$ and $w_i(r) = 0$ for all $i \neq y$ and $y_o(r) = 1$ and $y_i(r) = 0$ for all $i \neq o$. Therefore $p(r, z) \neq 0$ implies that for all polynomials $c_x(z)$, $c_y(z)$, $c_o(z)$, it is $c_x(z)c_y(z) \neq c_o(z)$, a contradiction.

($\Leftarrow$) Suppose $\tau(x)$ divides $p(x, z)$. Then $p(r, z) = 0$ for all multiplication gates r. By the definition of $v_i(x)$, $w_i(x)$, $y_i(x)$, the $c_1(z), c_2(z), \ldots, c_m(z)$ are correct assignments on the circuit wires. ■

We next give an efficient SNARK construction for polynomial circuits based on the above QPP. Recall that a polynomial circuit $\mathcal{F}$ has d multiplication gates and m wires, the wires $1, \ldots, N$ occupy inputs and outputs and set $I_m = \{N + 1, \ldots, m\}$ represents the internal wires, where $|I_m| \leq 3d$. Also, we denote with n_i the degree of polynomial on wire i and we set n to be an upper bound on the degrees of the polynomials on $\mathcal{F}$'s wires.

3.1 Intuition of Construction

The SNARK construction that we present works as follows. First, the key generation algorithm KeyGen produces a "commitment" to the polynomial circuit $\mathcal{F}$ by outputting elements that relate to the internal set of wires I_m of the QPP $\mathcal{Q} = (\mathcal{V}, \mathcal{W}, \mathcal{Y}, \tau(x))$ as the public key. These elements encode bivariate polynomials in the exponent, evaluated at randomly chosen points t and s, to accommodate for the fact that circuit $\mathcal{F}$ encodes operations over univariate polynomials and not just arithmetic values (as is the case with [14]).

As was described in the previous section, for the prover to prove that an assignment $c_1(z), c_2(z), \ldots, c_N(z)$ of polynomials on input/output wires is valid, it suffices to prove there exist polynomials $c_{N+1}(z), \ldots, c_m(z)$ corresponding to assignments on the internal wires, such that the polynomial $p(x, z)$ from Relation 3.1 has roots $r_1, r_2, \ldots, r_d$. To prove this, the prover first "solves" the circuit and computes the polynomials $c_1(z), c_2(z), \ldots, c_m(z)$ that correspond to the correct assignments on the wires. Then he uses these polynomials and the public evaluation key (i.e., the circuit "commitment") to compute the following three types of terms (which comprise the actual proof). The detailed computation of these values is described in Section 3.2.

- **Extractability terms.** These terms declare three polynomials in the exponent, namely polynomials $\sum_{k=N+1}^{m} c_k(z)v_k(x)$, $\sum_{k=N+1}^{m} c_k(z)w_k(x)$, and $\sum_{k=N+1}^{m} c_k(z)y_k(x)$. These polynomials correspond to the internal wires since the verifier can fill in the parts for the input and output wires.

  The above terms are engineered to allow extractability using a knowledge assumption. In particular, given these terms, there exists a polynomial-time extractor that can, with overwhelming probability, recover the assignment $c_{N+1}(z), \ldots, c_m(z)$ on internal wires. This proves the existence of $c_{N+1}(z), \ldots, c_m(z)$.

- **Consistency check terms.** Extraction is done separately for terms related to the three polynomials $\sum_{k=N+1}^{m} c_k(z)v_k(x)$, $\sum_{k=N+1}^{m} c_k(z)w_k(x)$, and $\sum_{k=N+1}^{m} c_k(z)y_k(x)$. We therefore require a set of consistency check terms to ensure that the extracted $c_{N+1}(z), \ldots, c_m(z)$ polynomials are consistent for the above $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$ terms—otherwise, the same wire can have ambiguous assignments.

- **Divisibility check term.** Finally, the divisibility check term is to ensure that the above divisibility check corresponding to relation $p(x, z) =$

$h(x,z)\tau(x)$, holds for the polynomial

$$\left(\sum_{k=1}^{m} c_k(z)v_k(x)\right)\left(\sum_{k=1}^{m} c_k(z)w_k(x)\right) - \left(\sum_{k=1}^{m} c_k(z)y_k(x)\right)$$

declared earlier by the extractability terms.

3.2 Concrete Construction

We now give the algorithms of our SNARK construction, (following Definition 3). In comparison with the QSP and QAP constructions [14, 23], one difficulty arises in our setting when working with polynomials on wires. In essence, to express a polynomial $c_k(z)$ on a wire in our construction, we evaluate the polynomial at a committed point $z = t$. In existing QSP and QAP constructions, the prover knows the cleartext value on each wire when constructing the proof. However, in our setting, the prover does not know what t is, and hence cannot directly evaluate the polynomials $c_k(z)$'s on each wire. In fact, security would be broken if the prover knew the value of the polynomials at $z = t$.

To overcome this problem, we have to include more elements in the evaluation key which will contain exponent powers of the variable t (see the evaluation key below). In this way, the prover will be able to evaluate $c_k(t)$ in the exponent, without ever learning the value t. We now give the algorithms:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathcal{F}, 1^k)$: Let $\mathcal{F}$ be a polynomial circuit. Build the corresponding QPP $\mathcal{Q} = (\mathcal{V}, \mathcal{W}, \mathcal{Y}, \tau(x))$ as above. Let e be a non-trivial bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, and let g be a generator of $\mathbb{G}$. $\mathbb{G}$ and $\mathbb{G}_T$ have prime order p. Pick $s, t, r_v, r_w, \alpha_v, \alpha_w, \alpha_y, \beta, \gamma$ from $\mathbb{Z}_p$ and set $r_y = r_v r_w$ and $g_v = g^{r_v}$, $g_w = g^{r_w}$ and $g_y = g^{r_y}$. The public evaluation key $\mathsf{EK}_\mathcal{F}$ is

1. $\{g_v^{t^i v_k(s)}, g_w^{t^i w_k(s)}, g_y^{t^i y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
2. $\{g_v^{t^i \alpha_v v_k(s)}, g_w^{t^i \alpha_w w_k(s)}, g_y^{t^i \alpha_y y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
3. $\{g_v^{t^i \beta \cdot v_k(s)} g_w^{t^i \beta \cdot w_k(s)} g_y^{t^i \beta \cdot y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
4. $\{g^{t^i s^j}\}_{(i,j)\in[2n]\times[d]}$.

The verification key $\mathsf{VK}_\mathcal{F}$ consists of the values

$$g, g^{\alpha_v}, g^{\alpha_w}, g^{\alpha_y}, g^{\gamma}, g^{\beta\gamma} g_y^{t(s)}$$

and the set $\{g_v^{t^i v_k(s)}, g_w^{t^i w_k(s)}, g_y^{t^i y_k(s)}\}_{(i,k)\in[n]\times[N]}$. Note $\mathsf{VK}_\mathcal{F}$ and $\mathsf{EK}_\mathcal{F}$ are the public key pk of the SNARK. Our SNARK is publicly verifiable, hence $\mathsf{sk} = \mathsf{pk}$.

$\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$: The input x contains input polynomials u and output polynomials y and the witness w (which contains assignments of polynomials on the internal wires). Let $c_k(z)$ be the polynomials on the circuit's wires such that $y = \mathcal{F}(u, w)$. Let $h(x, z)$ be the polynomial such that $p(x, z) = h(x, z) \cdot \tau(x)$. The proof is computed as follows:

1. *(Extractability terms)* $g_v^{v_m(s,t)}$, $g_w^{w_m(s,t)}$, $g_y^{y_m(s,t)}$, $g_v^{\alpha_v v_m(s,t)}$, $g_w^{\alpha_w w_m(s,t)}$, $g_y^{\alpha_y y_m(s,t)}$.
2. *(Consistency check term)* $g_v^{\beta \cdot v_m(s,t)} g_w^{\beta \cdot w_m(s,t)} g_y^{\beta \cdot y_m(s,t)}$.
3. *(Divisibility check term)* $g^{h(s,t)}$, where

(a) $v_m(x, z) = \sum_{k\in I_m} c_k(z)v_k(x)$;
(b) $w_m(x, z) = \sum_{k\in I_m} c_k(z)w_k(x)$; and
(c) $y_m(x, z) = \sum_{k\in I_m} c_k(z)y_k(x)$. Note that the term $g_v^{\beta \cdot v_m(s,t)} g_w^{\beta \cdot w_m(s,t)} g_y^{\beta \cdot y_m(s,t)}$ can be computed from public key terms $\{g_v^{t^i \beta \cdot v_k(s)} g_w^{t^i \beta \cdot v_k(s)} g_y^{t^i \beta \cdot y_k(s)}\}_{(i,k)\in[n]\times I_m}$.

$\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{pk}, x, \pi)$: Parse the proof π as

1. $\gamma_v, \gamma_w, \gamma_y, \kappa_v, \kappa_w, \kappa_y$.
2. Λ.
3. γ_h.

First, verify all three α terms: $e(\gamma_v, g^{\alpha_v}) \stackrel{?}{=} e(\kappa_v, g) \wedge e(\gamma_w, g^{\alpha_w}) \stackrel{?}{=} e(\kappa_w, g) \wedge e(\gamma_y, g^{\alpha_y}) \stackrel{?}{=} e(\kappa_y, g)$. Then verify the divisibility requirement:

$$e(\lambda_v \cdot \gamma_v, \lambda_w \cdot \gamma_w)/e(\lambda_y \cdot \gamma_y, g) \stackrel{?}{=} e(\gamma_h, g^{\tau(s)}),$$

where $\lambda_v = g^{\sum_{k\in[N]} c_k(t)v_k(s)}$, $\lambda_w = g^{\sum_{k\in[N]} c_k(t)w_k(s)}$, $\lambda_y = g^{\sum_{k\in[N]} c_k(t)y_k(s)}$. Finally verify the β term:

$$e(\gamma_v \cdot \gamma_w \cdot \gamma_y, g^{\beta\gamma}) \stackrel{?}{=} e(\Lambda, g^{\gamma}).$$

3.3 Asymptotic Complexity and Security

In this section we analyze the asymptotic complexity of our SNARK construction for polynomial circuits. We also state the security of our scheme.

KeyGen: It is easy to see that the computation time of KeyGen is $O(n|I_m| + nd + nN) = O(dn)$.

Prove: Let T be the time required to compute the polynomials $c_i(z)$ for $i = 1, \ldots, m$ and let n_i be the degree of the polynomial $c_i(z)$ for $i = 1, \ldots, m$. The computation of each $g^{c_i(z)v_i(x)}$ (similarly for $g^{c_i(z)w_i(x)}$ and $g^{c_i(z)y_i(x)}$) for $i \in I_m$ takes $O(n_i)$ time (specifically, $7 \cdot \sum n_i$ exponentiations are required to compute all the

proof), since one operation per coefficient of $c_i(z)$ is required. Then multiplication of $|I_m|$ terms is required. Therefore the total time required is

$$O\left(T + \sum_{i \in I_m} n_i + |I_m|\right) = O\left(T + d\nu\right) ,$$

where $\nu = \max_{i=1,\ldots,m}\{n_i\}$ is the maximum degree of the polynomials over the wires and since $|I_m| \leq 3d$. To compute $p(x,z)$, first the degree d polynomials $v_i(x), w_i(x), y_i(x)$ for $i = 1, ..., m$ are parsed in time $O(dm)$. Then $p(x,z)$ is computed according to Equation 1; each summation term is computed in time $O(d\nu)$ with naive bivariate polynomial multiplication and then they are summed for total complexity of $O(md\nu)$. For the division, note that $p(x,z)$ has maximum degree in z equal to 2ν and maximum degree in x equal to $2d$. To do the division, we apply "the change of variable trick". We set $z = x^{2\times(2d)+1}$ and therefore turn $p(x,z)$ into a polynomial of one variable x, namely the polynomial $p(x, x^{2\times(2d)+1})$. Therefore the dividend now has maximum degree $2\nu(4d+1)+2d$ while the divisor has still degree d. By using FFT, we can do such division in $O(d\nu \log(d\nu))$ time. Therefore the total time for Prove is $O\left(T + d\nu \log(d\nu) + md\nu\right)$.

Verify: The computation of each element $g^{c_i(z)v_i(x)}$ (resp. for $g^{c_i(z)w_i(x)}$ and $g^{c_i(z)y_i(x)}$) for $i = 1, \ldots, N$ takes $O(n_i)$ time, since one operation per coefficient of $c_i(z)$ is required. Then multiplication of N terms is required. Hence, the total time required is $O(\sum_{i\in[N]} n_i)$, proportional to the size of the input and output.

We now have the following result. The involved assumptions can be found in Appendix 6.1 and we provide its proof of security in the full version of our paper [20].

Theorem 1 (Security of the SNARK for $\mathcal{F}$) *Let $\mathcal{F}$ be a polynomial circuit with d multiplication gates. Let n be an upper bound on the degrees of the polynomials on the wires of $\mathcal{F}$ and let $q = 4d + 4$. The construction above is a SNARK under the $2(n+1)q$-PKE, the $(n+1)q$-PDH and the $2(n+1)q$-SDH assumptions.*

4 Efficient SNARKs for Set Circuits

In this section, we show how to use the SNARK construction for polynomial circuits from the previous section to build a SNARK for set circuits.

We first define a mapping from sets to polynomials (see Definition 5– such representation was also used in prior work, e.g., the work of Kissner and Song [18]). Then we express the correctness of the operations between two sets as constraints between the polynomials produced from this mapping (e.g., see Lemma 2). For a set operation to be correct, these constraints must be satisfied simultaneously. To capture that, we represent all these constraints with a circuit with loops, where a wire can participate in more than one constraint (see Figure 3).

4.1 Expressing Sets with Polynomials

We first show how to represent sets and set operations with polynomials and polynomial operations. This representation is key for achieving *input-specific* time, since we can represent a set with a polynomial evaluated at a random point (regardless of its cardinality). Given a set, we define its *characteristic polynomial*.

Definition 5 (Characteristic polynomial) *Let A be a set of elements $\{a_1, a_2, \ldots, a_n\}$ in $\mathbb{F}$. We define its characteristic polynomial as $\mathsf{A}(z) = (z + a_1) \ldots (z + a_n)$.*

We now show the relations between set operations and polynomial operations. Note that similar relations were used by Papamanthou et al. [22] in prior work.

Lemma 2 (Intersection constraints) *Let A, B and I be three sets of elements in $\mathbb{F}$. Then $\mathsf{I} = \mathsf{A} \cap \mathsf{B}$ iff there exist polynomials $\alpha(z)$, $\beta(z)$, $\gamma(z)$ and $\delta(z)$ such that*

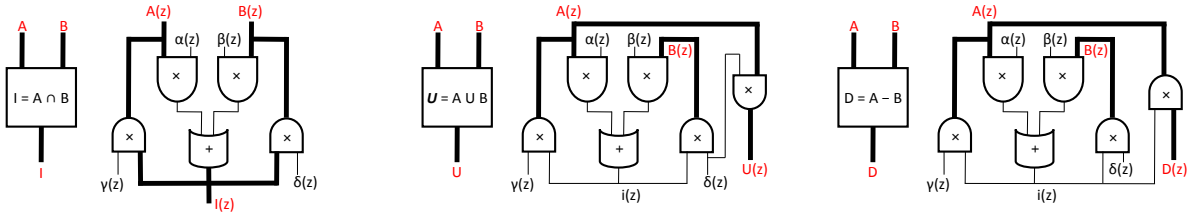
1. $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = \mathsf{I}(z)$.
2. $\gamma(z)\mathsf{I}(z) = \mathsf{A}(z)$.
3. $\delta(z)\mathsf{I}(z) = \mathsf{B}(z)$.

Proof: ($\Rightarrow$) If $\mathsf{I} = \mathsf{A} \cap \mathsf{B}$, it follows that (i) the great common divisor of polynomials $\mathsf{A}(z)$ and $\mathsf{B}(z)$ is $\mathsf{I}(z)$, therefore, by Bézout's identity, there exist polynomials $\alpha(z)$ and $\beta(z)$ such that (i) $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = \mathsf{I}(z)$; (ii) $\mathsf{I}(z)$ divides $\mathsf{A}(z)$ and $\mathsf{B}(z)$, therefore there exist polynomials $\gamma(z)$ and $\delta(z)$ such that $\gamma(z)\mathsf{I}(z) = \mathsf{A}(z)$ and $\delta(z)\mathsf{I}(z) = \mathsf{B}(z)$.

($\Leftarrow$) Let A, B and I be sets. Suppose there exist polynomials $\alpha(z)$, $\beta(z)$, $\gamma(z)$ and $\delta(z)$ such that (1), (2) and (3) are true. By replacing (2) and (3) into (1), we get that $\alpha(z)$ and $\beta(z)$ do not have any common factor, therefore $\mathsf{I}(z)$ is the greatest common divisor of $\mathsf{A}(z)$ and $\mathsf{B}(z)$ and therefore $\mathsf{A} \cap \mathsf{B} = \mathsf{I}$. ■

Corollary 1 (Union constraints) *Let A, B and U be three sets of elements in $\mathbb{F}$. Then $\mathsf{U} = \mathsf{A} \cup \mathsf{B}$ iff $\exists$ polynomials $i(z)$, $\alpha(z)$, $\beta(z)$, $\gamma(z)$ and $\delta(z)$ such that*

1. $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = i(z)$.
2. $\gamma(z)i(z) = \mathsf{A}(z)$.
3. $\delta(z)i(z) = \mathsf{B}(z)$.
4. $\delta(z)\mathsf{A}(z) = \mathsf{U}(z)$.



(a) (b) (c)

Figure 3: Set circuits for intersection (a), union (b) and difference (c) expressed as polynomial circuits with loops using Lemma 2, Corollary 1 and Corollary 2.

Corollary 2 (Difference constraints) *Let* A, B *and* D *be three sets of elements in* $\mathbb{F}$. *Then* $\mathsf{D} = \mathsf{A} - \mathsf{B}$ *iff* $\exists$ *polynomials* $i(z)$, $\alpha(z)$, $\beta(z)$, $\gamma(z)$ *and* $\delta(z)$ *such that*

1. $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = i(z)$.
2. $\mathsf{D}(z)i(z) = \mathsf{A}(z)$.
3. $\delta(z)i(z) = \mathsf{B}(z)$.

4.2 From Set to Polynomial Circuits

Polynomial circuits with loops. To compile a set circuit into a circuit on polynomials, we need to check that the constraints in Lemma 2 and Corollaries 1 and 2 simultaneously satisfy for all intersection, union, and set difference gates respectively. Doing this in a straightforward manner seems to require implementing a Boolean AND gate using polynomial algebra, which introduces an unnecessary representation overhead.

We use a simple idea to avoid this issue, by introducing polynomial circuits with *loops*. This means that the circuit's wires, following the direction of evaluation, can contain loops, as shown in Figure 3. When a circuit contains loops, we require that there exist an assignment for the wires such that every gate's inputs and output are consistent. It is not hard to see that we can build a QPP for a polynomial circuit with loops.

From set circuits to polynomial circuits. Suppose we have a set circuit $\mathcal{C}$, as in Definition 1. We can compile $\mathcal{C}$ into a polynomial circuit with loops $\mathcal{F}$ as follows:

1. Replace every intersection gate g_I with the circuit of Figure 3(a), which implements the constraints in Lemma 2. Note that 6 additional wires per intersection gate are introduced during this compilation, 4 of which are free wires. Also, for each intersection gate, 4 polynomial multiplication gates are added.
2. Replace every union gate g_U of $\mathcal{C}$ with the circuit of Figure 3(b), which implements the set of constraints in Corollary 1. Note that 7 additional wires per union gate are introduced during this compilation, 3 of which are free wires. Also, for each union gate, 5 polynomial multiplication gates are added.
3. Replace every difference gate g_D of $\mathcal{C}$ with the circuit of Figure 3(c), which implements the set of constraints in Corollary 2. Note that 7 additional wires per union gate are introduced during this compilation, 3 of which are free wires. Also, for each difference gate, 5 polynomial multiplication gates are added.

4.3 Asymptotic Complexity and Security

Let $\mathcal{C}$ be a set circuit with d gates (out of which d_1 are intersection gates and d_2 are union and difference gates) and N inputs and outputs. After compiling $\mathcal{C}$ into an polynomial circuit with loops, we end up with a circuit $\mathcal{F}$ with $4d_1 + 5d_2$ multiplication gates since each intersection introduces 4 multiplication gates and each union or difference introduces 5 multiplication gates.

Therefore, a SNARK for set circuits with $d = d_1 + d_2$ gates can be derived from a SNARK for polynomial circuits with $4d_1 + 5d_2$ multiplication gates. Note that the complexity of Prove for the SNARK for set circuits is $O(d\nu \log^2 \nu \log\log \nu)$ because the prover runs the extended Euclidean algorithm to compute the polynomials on the free wires, which takes $O(t \log^2 t \log\log t)$ time, for t-degree polynomials as inputs.

Theorem 2 (Security of the SNARK for $\mathcal{C}$) *Let* $\mathcal{C}$ *be a set circuit that has* d *total gates and* N *total inputs and outputs. Let* n *be an upper bound on the cardinalities of the sets on the wires of* $\mathcal{C}$ *and let* $q = 16d_1 + 20d_2 + 4$, *where* d_1 *is the number of intersection gates and* d_2 *is the number of union and difference gates* ($d = d_1 + d_2$). *The construction above is a SNARK for the set circuit* $\mathcal{C}$ *under the* $2(n+1)q$-PKE, *the* $(n+1)q$-PDH *and the* $2(n+1)q$-SDH *assumptions.*

We note here that there do exist known SNARK constructions for languages in NP that have excellent asymptotic behavior and are *input-specific*, e.g., the work of Bitansky et al. [6], based on recursive proof composition.

Therefore, in theory, our SNARK asymptotics are the same with the ones by Bitansky et al. [6] (when applied to the case of set operations). However, the concrete overhead of such techniques remains high; in fact, for most functionalities it is hard to deduce the involved constants. In comparison, with our approach, we can always deduce an upper bound on the number of necessary operations involved. We give a tight complexity analysis of our approach in the full version of our paper [20].

4.4 Handling More Expressive Circuits

As discussed in the introduction, by moving from QAPs to QPPs our scheme is not losing anything in expressiveness. So far we explicitly discussed the design of efficient set circuits that only consist of set gates. Ideally, we want to be able to efficiently accommodate "hybrid" circuits that consist both of set and arithmetic operations in an optimally tailored approach.

In this section we show how, by constructing a *split* gate (and a *merge* gate) that upon input a set A outputs its elements a_i, we gain some "backwards compatibility" with respect to QAPs. In particular, this allows us to compute on the set elements themselves, e.g., performing `MAX` or `COUNT`. Also, using techniques described by Parno et al. [23], one can go one step below in the representation hierarchy and represent a_i's in binary form which yields, for example, more efficient comparison operations.

Hence we produce a complete toolkit that a delegating client can use for a general purpose computation, in a way that allows it both to be more efficient for the part corresponding to set operations and at the same time perform arithmetic and bit operations optimally, choosing different levels of abstraction for different parts of the circuit.

Zero-degree assertion gate. Arithmetic values can be naturally interpreted as zero-degree polynomials. Since we want to securely accommodate both polynomials and arithmetic values in our circuit, we need to construct a gate that will constrain the values of some wires to arithmetic values. For example, we need to assure that the outputs of a split gate are indeed numbers (and not higher degree polynomials).

Lemma 3 (Zero-degree constraints) *Let $p(z)$ be a univariate polynomial in $\mathbb{F}[z]$. The degree of $p(z)$ is 0 iff $\exists$ polynomial $q(z)$ in $\mathbb{F}[z]$ such that $p(z)q(z) = 1$.*

Proof: ($\Rightarrow$) Every zero-degree polynomial $q(z) \in \mathbb{F}[z]$ also belongs in $\mathbb{F}$. Since every element in $\mathbb{F}$ has an inverse, the claim follows. ($\Leftarrow$) Assume now that $p(z)q(z) = 1$. Since polynomial 1 is of degree 0, $p(z)q(z)$ must also be of degree 0. By polynomial multiplication, we know that $p(z)q(z)$ has degree $deg(p(z)) + deg(q(z))$. Hence $deg(p(z)) = deg(q(z)) = 0$. ■

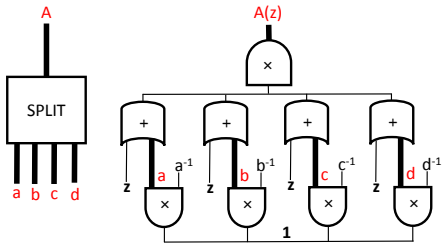


Figure 4: Implementation of a split gate for the set $\mathsf{A} = \{a, b, c, d\}$. The elements z and 1 on the wires are hard-coded in the circuit during setup. All other polynomials on the wires are computed by the prover.

This simple gate consists of a multiplication gate between polynomial $p(z)$ and an auxiliary input $q(z)$ computed by the server and the output is set to the (hard-coded) polynomial 1. If the input is indeed a zero-degree polynomial, by the above Lemma, $q(z)$ is easily computable by the server (an inverse computation in $\mathbb{F}$).

Split gate. A split gate, depicted in Figure 4, operates as follows. On input a wire with value $\mathsf{A}(z)$, it outputs n wires with the individual elements a_i. First, each of the wires carrying a_i is connected to a degree-zero assertion gate. This will make sure that these wires carry arithmetic values. Second, each of these wires is used as an input to an addition gate, with the other input being the degree-one polynomial z. Then the outputs of all the addition gates are multiplied together and the output of the multiplication is connected to the wire carrying $\mathsf{A}(z)$.

Split gate with variable number of outputs. In the above we assumed that the split gate has a fixed number of outputs, n. However, the number of outputs can vary. To accommodate this, we assume that n is an upper bound on the number of outputs of a split gate. Now, for each of the n output wires, we introduce an indicator variable ν_i (picked by the prover) such that if $\nu_i = 1$, this output wire is occupied and carries an arithmetic value, otherwise $\nu_i = 0$. Then, in the split gate of Figure 4, instead of $\prod_{i=1}^{n}(z + a_i)$ we compute $\prod_{i=1}^{n}[\nu_i(z + a_i) + (1 - \nu_i)]$. Note here that an additional restriction we need to impose is that $\nu_i \in \{0, 1\}$. Fortunately this can be checked very easily by adding one self-multiplication gate and a loop wire for each value that enforces the condition $\nu_i \cdot \nu_i = \nu_i$ that clearly holds iff $\nu_i = 0$ or 1.

Cardinality gate. One immediate side-effect of our construction for split gates with variable number of outputs, is that it indicates a way to construct another very important type of gate, namely a *cardinality* gate. Imagine for example a computation where the requested output is not a set but only its cardinality (e.g., a `COUNT` SQL-query or the Jaccard similarity index). A cardinality gate is implemented exactly like a split gate, however it only has a

single output wire that is computed as $\sum_i \nu_i$, using $n-1$ addition gates over the ν_i wires.

Merge gate. Finally, the *merge* gate upon input n wires carrying numerical values a_i, outputs a single wire that carries them as a set (i.e., its characteristic polynomial). The construction is similar to that of the split gate, only in reverse order. First input wires are tested to verify they are of degree 0, with n zero-degree assertion gates. Then, these wires are used as input for union gates, taken in pairs, in an iterative manner (imagine a binary tree of unions with n leaves and the output set at the root).

5 Evaluation

We now present the evaluation of TRUESET comparing its performance with Pinocchio [23], which is the state-of-the-art general VC scheme (already reducing computation time by orders-of-magnitude when compared with previous implementations). We also considered alternative candidates for comparison such as Pantry [9] which is specialized for stateful computations. Pantry is theoretically more efficient than Pinocchio, as it can support a RAM-based $O(n)$-time algorithm for computing set intersection (i.e., when the input sets are sorted), instead of the circuit-based $O(n \log^2 n)$ or $O(n^2)$ algorithms that Pinocchio supports. However, evaluation showed that Pantry requires considerable proof construction time, even for simple memory-based operations (e.g., 92 seconds for a single verifiable *put* operation in a memory of 8192 addresses), hence we chose to compare only with Pinocchio.

In our experiments, we analyze the performance of TRUESET both for the case of a single set operation and multiple set operations. We begin by presenting the details of our implementation and the evaluation environment and then we present the performance results.

5.1 Implementation

We built TRUESET by extending Pinocchio's C++ implementation so that it can handle set circuits, with the special set gates that we propose. However, since the original implementation of Pinocchio used efficient libraries for pairing-based cryptography and field manipulation that are not available for public use (internal to Microsoft), the first step was to replace those libraries with available free libraries that have similar characteristics. In particular, we used the Number Theory Library (NTL) [27] along with the GNU Multi-Precision (GMP) library [15] for polynomial arithmetic, in addition to an efficient free library for ate-pairing over Barreto-Naehrig curves [4], in which the underlying BN curve is $y^2 = x^3 + 2$ over a 254-bit prime field $\mathbb{F}_p$ that maintains a 126 bit-level of security. As in Pinocchio, the size of the cryptographic proof produced by our implementation is typically equal to 288 bytes in all experiments regardless of the input or circuit sizes.

TRUESET's executable receives an input file describing a set circuit that contains one or more of the set gates described earlier. The executable compiles the circuit to a QPP in two stages. In the first stage, the set gates are transformed into their equivalent representation using polynomial multiplication and addition gates, as in Figures 3 and 4, and then the QPP is formed directly in the second stage by generating the roots, and calculating the V, W and Y polynomials.

Optimizations. For a fair comparison, we employ the same optimizations used for reducing the exponentiation overhead in Pinocchio's implementation. Concerning polynomial arithmetic, Pinocchio's implementation uses an FFT approach to reduce the polynomial multiplication costs. In our implementation, we use the NTL library, which already provides an efficient solution for polynomial arithmetic based on FFT [28].

In addition to the above, the following optimizations were found to be very useful when the number of set gates is high, or when the set split gate is being used.
1) For key generation, we reduce the generated key size by considering the maximum polynomial degree that can appear on each wire, instead of assuming a global upper bound on the polynomial degree for all wires (as described in previous sections). This can be calculated by assuming a maximum cardinality of the sets on the input wires, and then iterating over the circuit wires to set the maximum degree per wire in the worst case, e.g. the sum of the worst case cardinalities of the input sets for the output of a union gate, and the smaller for intersections.
2) The NTL library does not provide direct support for bivariate polynomial operations, needed to calculate $h(x,z)$ through division of $p(x,z)$ by $\tau(x)$. Hence, instead of doing a naive $O(n^2)$ polynomial division, we apply the change-of-variable trick discussed in Section 3.3 to transform bivariate polynomials into univariate ones that can be handled efficiently with NTL FFT operations.
3) Finally, calculation of the coefficients of the characteristic polynomial corresponding to the output is done by the prover and not by the verifier. The verifier then verifies that the set elements of the output (i.e., the roots of the characteristic polynomials) match the polynomial (expressed in coefficients) returned by the server. This can be efficiently done through a randomized check—see algorithm certify() from [22]. We specify that this slightly increases the communication bandwidth (the server effectively sends the output set twice, in two different encodings) but we consider this an acceptable overhead (This can be avoided by having the client perform the interpolation himself, increasing the verification time). It can also be noted that the input polynomial coefficients computa-

tion can be outsourced similarly to the server side, if the client does not have them computed already.

5.2 Experiments Setup

We now provide a comparison between TRUESET's approach and Pinocchio's approach based for set operations. For a fair comparison, we considered two different ways to construct the arithmetic circuits used by Pinocchio to verify the set operations:

- Pairwise comparison-based, which is the naive approach for performing set operations. This requires $O(n^2)$ equality comparisons.
- Sorting network-based, in which the input sets are merged and sorted first using and odd-even merge-sort network [19]. Then a check for duplicate consecutive elements is applied to include/remove repeated elements, according to the query being executed. This requires $O(n \log^2 n)$ comparator gates, and $O(n)$ equality gates.

Although the second approach is asymptotically more efficient, when translated to Pinocchio's circuits it results in numerous multiplication gates. This is due to the k-bits split gates needed to perform comparison operations, resulting into great overhead in the key generation and proof computation stages. For a k-bit possible input value, this split gate needs k multiplication constraints to constrain each bit wire to be either 0 or 1. (It should be noted that these gates translate a wire into its bit-level representation and they should not be confused with the split gates we introduce in this paper, which output the elements of a set as separate arithmetical values). On the other hand, the pairwise approach uses zero-equality gates to check for equality of elements. Each equality gate translates into only two multiplication gates, requiring only two roots.

For fairness purposes, different Pinocchio circuits were produced for each different input set cardinality we experiment with, as each wire in Pinocchio's circuits represents a single element. On the other hand, TRUESET can use the same circuit for different input cardinalities.

We consider two Pinocchio circuit implementations:

- MS Pinocchio: This is the executable built using efficient Microsoft internal libraries.
- NTL-ZM Pinocchio: This is a Pinocchio version built using exactly the same free libraries we used for our TRUESET implementation. This will help ensure having a fair comparison.

The experiments were conducted on a Lenovo IdeaPad Y580 Laptop. The executable used a single core of a 2.3 GHz Intel Core i7 with 8 GB of RAM. For the input sets, disjoint sets containing elements in $\mathbb{F}$ were assumed. For running time statistics, ten runs were collected for each data point, and the 95% confidence interval was calculated. Due to the scale of the figures, the confidence interval of the execution times (i.e., error bars) was too low to be visualized.

5.3 Single-Gate Circuit

In this subsection, we compare TRUESET and Pinocchio's protocols based on the verification of a single union operation that accepts two input sets of equal cardinalities. We study both the time overhead and the key sizes with respect to different input set cardinalities. Note that, experiments for higher input cardinalities in Pinocchio's case incur great memory overhead due to the large circuit size, therefore we were unable to even perform Pinocchio's for large input sizes.

Figure 5 shows the comparison between TRUESET's approach and Pinocchio's pairwise and sorting network approaches, versus the cardinality of each input set. The results show clearly that TRUESET outperforms both approaches in the key generation and proof computation stages by orders of magnitude, while maintaining the same verification time. Specifically, TRUESET outperforms Pinocchio in the prover's running time by *150x* when the input set cardinality is 2^8. This saving happens in both polynomial computations and exponentiation operations, as shown in Figure 5 (c). We also note that Pinocchio's pairwise comparison approach outperforms the sorting network approach due to the expensive split gates needed for comparisons in the sorting-network circuits, as discussed above, which results into a large constant affecting the performance at small cardinalities.

Considering evaluation and verification key sizes, Figure 5 also shows a comparison between TRUESET and Pinocchio under both the pairwise and sorting networks approaches. The figures demonstrate that TRUESET yields much smaller evaluation keys due to the more compact wire representation it employs (a single wire for a set as opposed to a wire per element), e.g., at an input set cardinality of 2^8, the saving is about 98%. It can also be noticed that the keys generated in Pinocchio using sorting networks are much larger than the ones generated in pairwise circuits, due to the use of the split gates. On the other hand, TRUESET and Pinocchio almost maintain the same verification key sizes, as the verification key mainly depends on the number of input elements in addition to the number of output elements in the worst case. (The verification key in TRUESET is *negligibly* more than the verification key of Pinocchio, due to an additional value that is needed to be verified per each input or output set. This is because an n-element set is represented by an n-degree polynomial which requires $n + 1$ coefficients.)

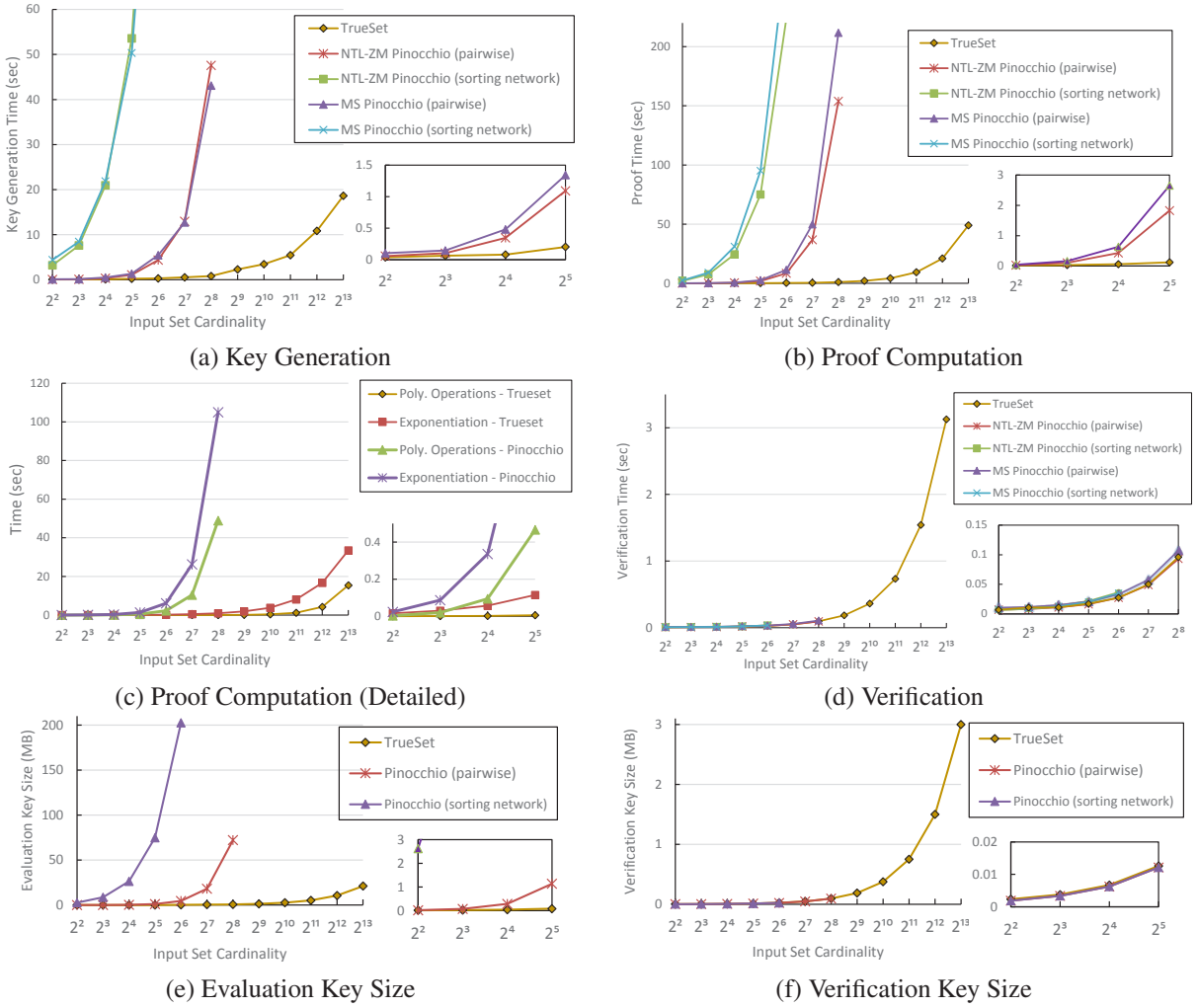


Figure 5: Comparison between TRUESET *and Pinocchio for the case of a single union gate. In the horizontal axis, we show the cardinality of* each *input set in logarithmic scale. (Note: Each time data point is the average of ten runs. The error bars were too small to be visualized). Subfigures (a), (b) and (d) show the comparison in terms of the key generation, proof computation and verification times, while (c) shows* TRUESET*'s prover's time in more detail compared to Pinocchio's prover in the case of pairwise comparison. Subfigures (e) and (f) show the compressed evaluation and verification key sizes (The cryptographic proof for all instances is 288 bytes).*

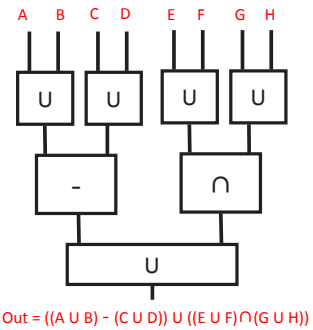


Figure 6: The multiple-gate circuit used for evaluation.

5.4 Multiple-Gate Circuit

We now compare TRUESET and Pinocchio's performance for a complex set circuit consisting of multiple set operations, illustrated in Figure 6. The circuit takes eight input sets of equal cardinalities, and outputs one set. We compare both the prover's overhead and the key sizes with respect to different input set cardinalities, but this time we consider only Pinocchio circuits based on pairwise comparisons, as the sorting network approach has much larger overhead for computation times and key sizes as shown in the previous subsection.

Figure 7 shows a comparison between TRUESET's approach and Pinocchio's approach. The results again

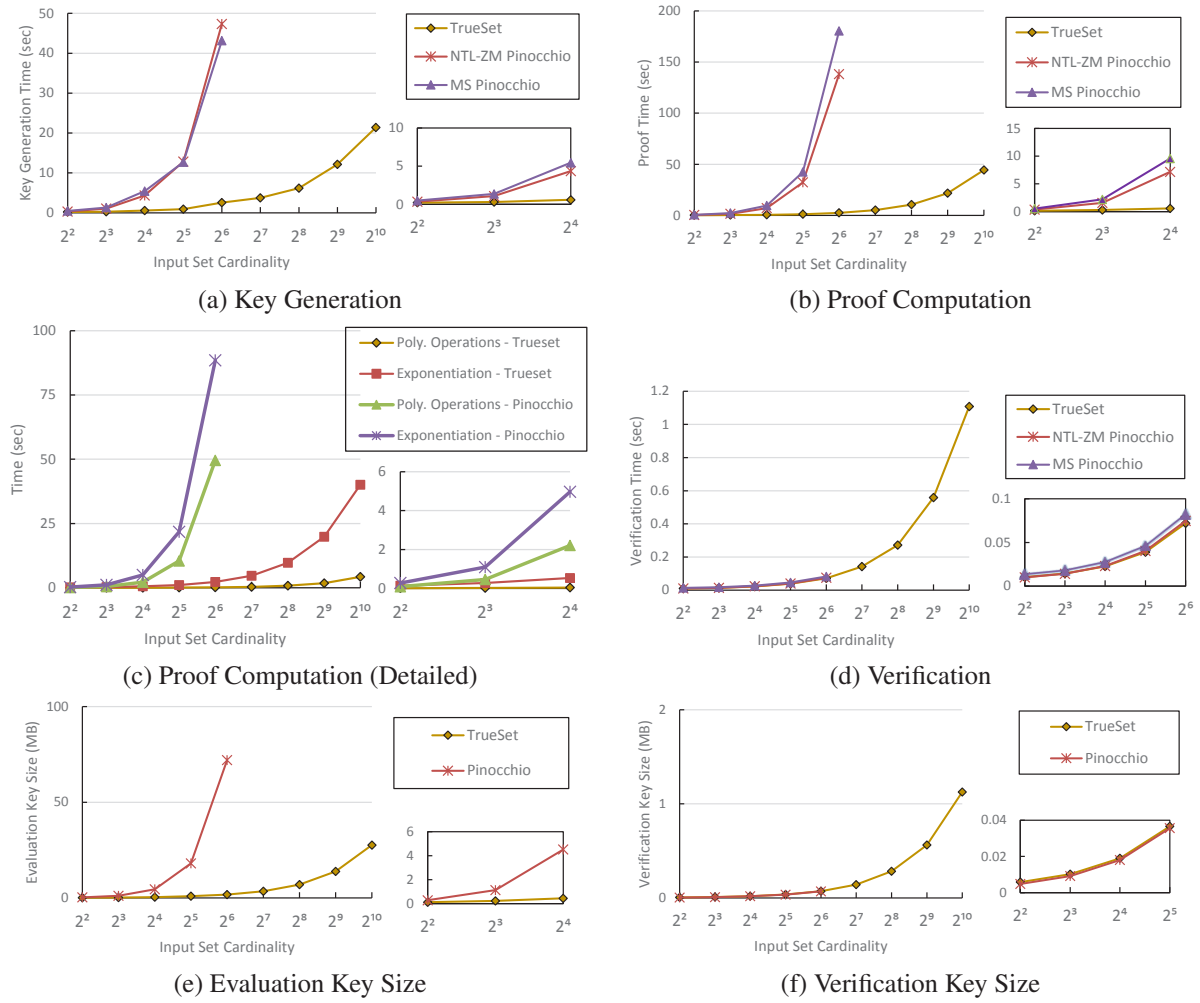


Figure 7: Comparison between TRUESET *and Pinocchio in the case of the multiple-gate circuit shown in Fig. 6, assuming the pair-wise comparison circuit for Pinocchio. In the horizontal axis, we show the cardinality of* each *input set in logarithmic scale. Subfigures (a), (b) and (d) show the comparison in terms of the key generation, proof computation and verification time, while (c) shows* TRUESET*'s prover's time in more detail compared to Pinocchio's prover time. Subfigures (e) and (f) show the compressed evaluation and verification key sizes (The cryptographic proof for all instances is 288 bytes).*

confirm that TRUESET greatly outperforms Pinocchio's elapsed time for key generation and proof computation, while maintaining the same verification time. In particular, for input set cardinality of 2^6, TRUESET's prover has a speedup of more than **50x**. In terms of key sizes, the figure confirms the observation that the evaluation key used by TRUESET is tiny compared to that of Pinocchio, e.g., 97% smaller when the input cardinality is 2^6.

5.5 Cardinality and Sum of Set Elements

Here, we evaluate TRUESET when a split gate is used to calculate the cardinality and sum for the output set of Figure 6. We compare that with Pinocchio's performance for the same functions. One important parameter that has to be defined for the split gate first is the maximum cardinality of the set it can support. This is needed for translating the split gate to the appropriate number of multiplication gates needed for verification. For example, a split gate added to the output of the circuit in Figure 6, will have to account for $4n$ set elements in the worst case, if n is the upper bound on the input set cardinalities.

Table 1 presents a comparison between TRUESET and Pinocchio in terms of the elapsed times in the three stages and the evaluation/verification key sizes, when the input set cardinality is 64. As the table shows, TRUESET can provide better performance in terms of the key generation and proof computation times (4x better proof computation time), in addition to a much smaller public evaluation key. It can be noted that, while there definitely exists a large improvement over Pinocchio, it is not as large as the one

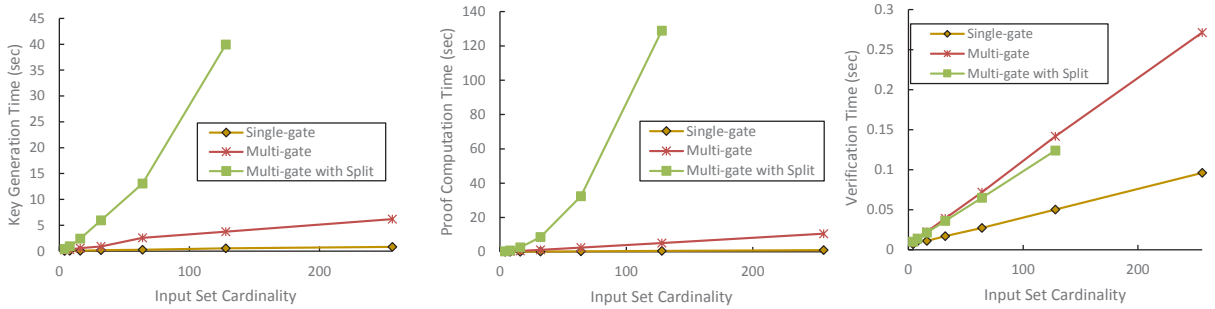


Figure 8: Summary of TRUESET *performance under all circuits in linear scale.*

	TRUESET	MS Pinocchio	NTL-ZM Pinocchio
Key Generation (sec)	13.07	43.03	47.39
Proof Computation (sec)	32.45	174.99	137.79
Verification (sec)	0.065	0.074	0.066
Evaluation Key (MB)	12.7	72.45	72.45
Verification Key (KB)	49.65	48.6	48.6

Table 1: Comparison between TRUESET *and Pinocchio on a circuit that computes* ***the cardinality and the sum*** *of the output set in the circuit in Figure 6, at input set cardinality of 64.*

exhibited for the previous single-gate and multiple-gate circuits. Overall, we found the split gate to be costlier than set gates since the multiplication gates introduced by the split gate increase proportionally with the number of the set elements it can support, whereas set gates are "oblivious" to the number of elements.

5.6 Discussion of Results

The evaluation of TRUESET for single-gate and multiple-gate circuits showed huge improvement for both key generation and proof computation time over Pinocchio. For example, for the single union case with 2^8-element input sets, a speed-up of 150x was obtained for the prover's time, while providing more than 98% saving in the evaluation key size. For a multiple-gate circuit comprised of seven set gates with eight input sets, each of 2^6 elements, a prover speed-up of more than 50x, and key size reduction of 97% were obtained.

As can be qualitatively inferred by our plots, these improvements in performance allow us to accommodate problem instances that are several times larger than what was considered achievable by previous works. TRUESET achieves the performance behavior that Pinocchio exhibits for sets of a few dozen elements, for sets that scale up to approximately 8000 elements, handling circuits with nearly 30x larger I/O size. Figure 8 summarizes the behavior of TRUESET for all circuits we experimented with, illustrating its performance for the three stages in linear scale. In all cases, the running time increases approximately linearly in the input size. The cost increases more abruptly when a split gate is introduced due to the added complexity discussed above. Improving the performance of the split gate is one possible direction for future work.

Remarks. We discuss here a few points related to the performance of our scheme.

Performance on Arithmetic Circuits. The presented evaluation covered the case of set circuits only, in which our construction outperformed arithmetic circuits verified using Pinocchio. Our construction can support typical arithmetic circuits as well, by assuming that the maximum polynomial degree on each wire is 0. In this case, our construction will reduce to Pinocchio's, however due to the bivariate polynomial operations, there will be more overhead in accommodating arithmetic circuits. For example, for an arithmetic circuit handling the multiplication of two 50x50 32-bit element matrices, the prover's time with TRUESET increased by 10% compared to Pinocchio.

Outsourced Sets. In the above, we assumed that the client possesses the input sets. However, it is common practice in cloud computing, to not only delegate computations but storage as well. In this case, the client initially outsources the sets to the server and then proceeds to issue set operation queries over them. This introduces the need for an additional mechanism to ensure the authenticity of the set elements used by the server. The full version of our paper [20] describes a modified protocol that handles this case using Merkle tree proofs.

Supporting multisets. Finally, it should be noted that the comparisons with Pinocchio above assumed proper sets only. In a setting that accommodates multiset operations (i.e., sets that allow repetition in elements), we expect TRUESET's performance to be much better, as it can naturally handle multiset cases without adding any modifications. On the other hand, Pinocchio multiset circuits are going to become more complex due to the added complexity of taking repetitions into account. For example, in intersection gates, it will not be enough to only check that two element are equal, but it will also be necessary to make sure that the matched element was not encountered before, introducing additional overhead.

Acknowledgments

We would like to thank David Evans and the anonymous reviewers for their invaluable comments and feedback.

References

[1] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *CCS*, pages 863–874, 2013.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, pages 90–108, 2013.

[3] E. Ben-Sasson, A. Chiesa, E. Tromer and M. Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. Cryptology ePrint Archive, Report 2013/879, 2013. `http://eprint.iacr.org/`.

[4] J.-L. Beuchat, J. E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over Barreto–Naehrig curves. In *Pairing*, pages 21–39. Springer, 2010.

[5] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.

[6] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120, 2013.

[7] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, pages 315–333, 2013.

[8] D. Boneh and X. Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *J. Cryptology*, 21(2), pages 149–177, 2008.

[9] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, pages 341–357, 2013.

[10] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *PKC*, pages 113–130, 2014.

[11] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In *CRYPTO*, pages 151–168, 2011.

[12] K.-M. Chung, Y. T. Kalai, and S. P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, pages 483–501, 2010.

[13] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[14] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, pages 626–645, 2013.

[15] T. Granlund and the GMP development team". *GMP: The GNU Multiple Precision Arithmetic Library*, 2006. Available at `http://gmplib.org/`.

[16] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, pages 321–340, 2010.

[17] P. Jaccard. *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz, 1901.

[18] L. Kissner and D. X. Song. Privacy-preserving set operations. In *CRYPTO*, pages 241–257, 2005.

[19] D. E. Knuth. *The art of computer programming*. Pearson Education, 2005.

[20] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Nearly practical verifiable set computations. Cryptology ePrint Archive, Report 2014/160, 2014. `http://eprint.iacr.org/2014/160`.

[21] S. Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[22] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

[23] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[24] B. Parno, M. Raykova, and V. Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, pages 422–439, 2012.

[25] S. T. V. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, pages 71–84, 2013.

[26] S. T. V. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[27] V. Shoup. *NTL: Number theory library*. Available at `http://www.shoup.net/ntl/`.

[28] V. Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20(4), pages 363–397, 1995.

[29] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 223–237, 2013.

6 Appendix

6.1 Computational Assumptions

Assumption 1 (q-PDH assumption [16]) *The q-power Diffie-Hellman (q-PDH) assumption holds for $\mathcal{G}$ if for all PPT $\mathcal{A}$ the following probability is negligible in k:*

$$\Pr\left[\begin{array}{l}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow\mathcal{G}(1^k);s\leftarrow\mathbb{Z}_p^*;\\ \mathbf{G}\leftarrow\left[g,g^s,\ldots,g^{s^q},g^{s^{q+2}},\ldots,g^{s^{2q}}\right];\\ \sigma\leftarrow(p,\mathbb{G},\mathbb{G}_T,e,\mathbf{G});\\ y\leftarrow\mathcal{A}(\sigma):y=g^{s^{q+1}}\end{array}\right].$$

Assumption 2 (q-PKE assumption [16]) *The q-power knowledge of exponent assumption holds for $\mathcal{G}$ if for all PPT $\mathcal{A}$ there exists a non-uniform PPT extractor $\chi_{\mathcal{A}}$ such that the following probability is negligible in k:*

$$\Pr\left[\begin{array}{l}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow\mathcal{G}(1^k);\{\alpha,s\}\leftarrow\mathbb{Z}_p^*;\\ \mathbf{G}\leftarrow\left[g,g^s,\ldots,g^{s^q},g^{\alpha},g^{\alpha s},\ldots,g^{\alpha s^q}\right];\\ \sigma\leftarrow(p,\mathbb{G},\mathbb{G}_T,e,\mathbf{G});\\ (c,\hat{c};a_0,a_1,\ldots,a_q)\leftarrow(\mathcal{A}||\chi_{\mathcal{A}})(\sigma,z):\\ \hat{c}=c^{\alpha}\wedge c\neq g^{\prod_{i=0}^{q}a_is^i}\end{array}\right],$$

for any auxiliary information $z\in\{0,1\}^{\mathsf{poly}(k)}$ that is generated independently of α. Note that $(y;z)\leftarrow(\mathcal{A}||\chi_{\mathcal{A}})(x)$ signifies that on input x, $\mathcal{A}$ outputs y, and that $\chi_{\mathcal{A}}$, given the same input x and $\mathcal{A}$'s random tape, produces z.

Assumption 3 (q-SDH assumption [8]) *The q-strong Diffie-Hellman (q-SDH) assumption holds for $\mathcal{G}$ if for all PPT $\mathcal{A}$ the following probability is negligible in k:*

$$\Pr\left[\begin{array}{l}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow\mathcal{G}(1^k);\{s\}\leftarrow\mathbb{Z}_p^*;\\ \sigma\leftarrow(p,\mathbb{G},\mathbb{G}_T,e,\mathbf{G}=\left[g,g^s,\ldots,g^{s^q}\right]);\\ (y,c)\leftarrow\mathcal{A}(\sigma):y=e(g,g)^{\frac{1}{s+c}}.\end{array}\right].$$

6.2 Succinct Non-Interactive Arguments of Knowledge (SNARKs)

Definition 6 (SNARK) *Algorithms* $(\mathsf{KeyGen},\mathsf{Prove},\mathsf{Verify})$ *give a* succinct non-interactive argument of knowledge (SNARK) *for an NP language L with corresponding NP relation R_L if:*

Completeness: *For all $x\in L$ with witness $w\in R_L(x)$, the following probability is negligible in k:*

$$\Pr\left[\mathsf{Verify}(\mathsf{sk},x,\pi)=0\;\middle|\;\begin{array}{c}(\mathsf{pk},\mathsf{sk})\leftarrow\mathsf{KeyGen}(1^k),\\ \pi\leftarrow\mathsf{Prove}(\mathsf{pk},x,w)\end{array}\right]$$

Adaptive soundness: *For any PPT algorithm $\mathcal{A}$, the following probability is negligible in k:*

$$\Pr\left[\begin{array}{c}\mathsf{Verify}(\mathsf{sk},x,\pi)=1\\ \wedge\,(x\notin L)\end{array}\;\middle|\;\begin{array}{c}(\mathsf{pk},\mathsf{sk})\leftarrow\mathsf{KeyGen}(1^k),\\ (x,\pi)\leftarrow\mathcal{A}(1^k,\mathsf{pk})\end{array}\right]$$

Succinctness: *The length of a proof is given by $|\pi|=\mathsf{poly}(k)\mathsf{poly}\log(|x|+|w|)$.*

Extractability: *For any poly-size prover* Prv*, there exists an extractor* Extract *such that for any statement x, auxiliary information μ, the following holds:*

$$\Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{sk})\leftarrow\mathsf{KeyGen}(1^k)\\ \pi\leftarrow\mathsf{Prv}(\mathsf{pk},x,\mu)\\ \mathsf{Verify}(\mathsf{sk},x,\pi)=1\\ \wedge\\ w\leftarrow\mathsf{Extract}(\mathsf{pk},\mathsf{sk},x,\pi)\\ w\notin R_L(x)\end{array}\right]=\mathsf{negl}(k).$$

Zero-knowledge: *There exists a simulator* Sim*, such that for any PPT adversary $\mathcal{A}$, the following holds:*

$$\Pr\left[\begin{array}{c}\mathsf{pk}\leftarrow\mathsf{KeyGen}(1^k);(x,w)\leftarrow\mathcal{A}(\mathsf{pk});\\ \pi\leftarrow\mathsf{Prove}(\mathsf{pk},x,w):(x,w)\in R_L\\ \textit{and}\;\mathcal{A}(\pi)=1\end{array}\right]$$

$$\simeq$$

$$\Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{state})\leftarrow\mathsf{Sim}(1^k);(x,w)\leftarrow\mathcal{A}(\mathsf{pk});\\ \pi\leftarrow\mathsf{Sim}(\mathsf{pk},x,\mathsf{state}):(x,w)\in R_L\\ \textit{and}\;\mathcal{A}(\pi)=1.\end{array}\right]$$

We say that a SNARK is *publicly verifiable* if $\mathsf{sk}=\mathsf{pk}$. In this case, proofs can be verified by anyone with pk. Otherwise, we call it a *secretly-verifiable* SNARK, in which case only the party with sk can verify.

Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture

Eli Ben-Sasson, *Technion*
Alessandro Chiesa, *MIT*
Eran Tromer, *Tel Aviv University*
Madars Virza, *MIT*

Abstract

We build a system that provides succinct non-interactive zero-knowledge proofs (*zk-SNARK*s) for program executions on a von Neumann RISC architecture. The system has two components: a cryptographic proof system for verifying satisfiability of arithmetic circuits, and a circuit generator to translate program executions to such circuits. Our design of both components improves in functionality and efficiency over prior work, as follows.

Our circuit generator is the first to be *universal*: it does not need to know the program, but only a bound on its running time. Moreover, the size of the output circuit depends *additively* (rather than multiplicatively) on program size, allowing verification of larger programs.

The cryptographic proof system improves proving and verification times, by leveraging new algorithms and a pairing library tailored to the protocol.

We evaluated our system for programs with up to 10,000 instructions, running for up to 32,000 machine steps, each of which can arbitrarily access random-access memory; and also demonstrated it executing programs that use *just-in-time compilation*. Our proofs are 230 bytes long at 80 bits of security, or 288 bytes long at 128 bits of security. Typical verification time is 5 ms, regardless of the original program's running time.

1 Introduction

1.1 Goal

Consider the setting where a client owns a public input x, a server owns a private input w, and the client wishes to learn $z := F(x,w)$ for a program F known to both parties. For instance, x may be a query, w a confidential database, and F the program that executes the query on the database.

Security. The client is concerned about *integrity* of computation: how can he ascertain that the server reports the correct output z? In contrast, the server is concerned about *confidentiality* of his own input: how can he prevent the client from learning information about w?

Cryptography offers a powerful tool to address these security concerns: *zero-knowledge proofs* [43]. The server, acting as the prover, attempts to convince the client, acting as the verifier, that the following NP statement is true: "there exists w such that $z = F(x,w)$". Indeed:

- The *soundness* property of the proof system guarantees that, if the NP statement is false, the prover cannot convince the verifier (with high probability). Thus, soundness addresses the client's integrity concern.
- The *zero-knowledge* property of the proof system guarantees that, if the NP statement is true, the prover can convince the verifier without leaking any information about w (beyond was is leaked by the output z). Thus, zero knowledge addresses the server's confidentiality.

Moreover, the client sometimes not only seeks soundness but also *proof of knowledge* [43, 11], which guarantees that, whenever he is convinced, not only can he deduce that a witness w exists, but also that the server *knows* one such witness. This stronger property is often necessary to security if F encodes cryptographic computations, and is satisfied by most zero-knowledge proof systems.

Efficiency. Besides the aforementioned security desiderata, many settings also call for *efficiency* desiderata. The client may be either unable or unwilling to engage in lengthy interactions with the server, or to perform large computations beyond the "bare minimum" of sending the input x and receiving the output z. For instance, the client may be a computationally-weak device with intermittent connectivity (e.g., a smartphone).

Thus, it is desirable for the proof to be *non-interactive* [25, 55, 23]: the server just send the claimed output $\tilde{z}$, along with a non-interactive proof string π that attests that $\tilde{z}$ is the correct output. Moreover, it is also desirable for the proof to be *succinct*: π has size $O_\lambda(1)$ and can be verified in time $O_\lambda(|F|+|x|+|z|)$, where $O_\lambda(\cdot)$ is some polynomial in a security parameter λ; in other words, π is very short and easy to verify (i.e., verification time does *not* depend on $|w|$, nor F's running time).

zk-SNARKs. A proof system achieving the above security and efficiency desiderata is called a (publicly-verifiable) *zero-knowledge Succinct Non-interactive ARgument of Knowledge* (zk-SNARK). zk-SNARK constructions can be applied to a wide range of security applications, provided these constructions deliver good enough *efficiency*, and support rich enough *functionality* (i.e., the class of programs F that is supported).

Remark 1.1. In the zero-knowledge setting above, the client does not have the server's input, and so cannot conduct the computation on his own. Hence, it is *not meaningful* to compare "efficiency of outsourced computation at the server" and "efficiency of native execution at the client", because the latter was never an option. Non-interactive zero-knowledge proofs (and zk-SNARKs) are useful regardless of *cross-over points*.

Our goal in this paper is to construct

a zk-SNARK implementation supporting executions on a universal von Neumann RISC machine.

1.2 Prior work

zk-SNARKs. Many works have obtained zk-SNARK constructions [45, 51, 38, 22, 56, 16, 52, 27]. Three of these [56, 16, 27] provide implementations, and thus we briefly recall them. Parno et al. [56] present two main contributions.

- A zk-SNARK, with essentially-optimal asymptotics, for arithmetic circuit satisfiability, based on *quadratic arithmetic programs* (QAPs) [38]. They accompany their construction with an implementation.
- A compiler that maps C programs with fixed memory accesses and bounded control flow (e.g., array accesses and loop iteration bounds are compile-time constants) into corresponding arithmetic circuits.

Ben-Sasson et al. [16] present three main contributions.

- Also a QAP-based zk-SNARK with essentially-optimal asymptotics for arithmetic circuit satisfiability, and a corresponding implementation. Their construction follows the linear-interactive proofs of [22].
- A simple RISC architecture, TinyRAM, along with a circuit generator for generating arithmetic circuits that verify correct execution of TinyRAM programs.
- A compiler that, given a C program, produces a corresponding TinyRAM program.

Finally, Braun et al. [27] re-implemented the protocol of [56] and combined it with a circuit generator that incorporates memory-checking techniques [24] to support random-access memory [14].

Outsourcing computation to powerful servers. Numerous works [63, 65, 66, 64, 32, 68, 71, 67, 27] seek to verifiably outsource computation to untrusted powerful servers, e.g., in order to make use of cheaper cycles or storage. (See Appendix A for a summary.) We stress that verifiable outsourcing of computations *is not our goal*. Rather, as mentioned, we study functionality and efficiency aspects of *non-interactive zero-knowledge proofs*, which are useful even when applied to relatively-small computations, and even with high overheads.

Compared to most protocols to outsource computations, known zk-SNARKs use "heavyweight" techniques, such as *probabilistically-checkable proofs* [6] and expensive pairing-based cryptography. The optimal choice of protocol, and whether it actually pays off compared to local *native execution*, are complex, computation-dependent questions [71], and we leave to future work the question of whether zk-SNARKs are useful for the goal of outsourcing computations.

1.3 Limitations of prior work

Recent work has made tremendous progress in taking zk-SNARKs from asymptotic theory into concrete implementations. Yet, known implementations suffer from several limitations.

Per-program key generation. As in any non-interactive zero-knowledge proof, a zk-SNARK requires a one-time trusted setup of public parameters: a *key generator* samples a proving key (used to generate proofs) and a verification key (used to check proofs). However, current zk-SNARK implementations [56, 16] require the setup phase to depend on the program F, which is *hard-coded* in the keys. Key generation is costly (quasilinear in F's runtime) and is thus difficult to amortize if conducted anew for each program. More importantly, per-program key generation requires, *for each new choice of program*, a trusted party's help.

Limited support for high-level languages. Known circuit generators have limited functionality or efficiency: (i) [56]'s circuit generator only supports programs without data dependencies, since memory accesses and loop iteration bounds cannot depend on a program's input; (ii) [27]'s circuit generator allows data-dependent memory accesses, but each such access requires expensive hashing to verify Merkle-tree authentication paths; (iii) [16]'s circuit generator supports arbitrary programs but its circuit size scales inefficiently with program size (namely, it has size $\Omega(\ell T)$ for ℓ-instruction T-step TinyRAM programs). Moreover, while there are techniques that mitigate some of the above limitations [72], these only apply in special cases, and not do address general data dependencies, a common occurrence in many programs.

Generic sub-algorithms. The aforementioned zk-SNARKs use several sub-algorithms, and in particular elliptic curves and pairings. Protocol-specific optimizations are a key ingredient in fast implementations of

pairing-based protocols [59], yet prior implementations only utilize off-the-shelf cryptographic libraries, and miss key optimization opportunities.

1.4 Results

We present two main contributions: a new circuit generator and a new zk-SNARK for circuits. These can be used independently, or combined to obtain an overall system.

1.4.1 A new circuit generator

We design and build a new circuit generator that incorporates the following two main improvements.

(1) Our circuit generator is *universal*: when given input bounds ℓ, n, T, it produces a circuit that can verify the execution of *any* program with $\leq \ell$ instructions, on *any* input of size $\leq n$, for $\leq T$ steps. Instead, all prior circuit generators [66, 64, 56, 16, 27] hardcoded the program in the circuit. Combined with a zk-SNARK for circuits (or any NP proof system for circuits), we achieve a notable conceptual advance: *once-and-for-all key generation* that allows verifying all programs up to a given size. This removes major issues in all prior systems: expensive per-program key generation, and the thorny issue of conducting it anew in a trusted way for every program.

Our circuit generator supports a universal machine that, like modern computers, follows the *von Neumann paradigm* (program and data lie in the same read/write address space). Concretely, it supports a von Neumann RISC architecture called vnTinyRAM, a modification of TinyRAM [17]. Thus, we also support programs leveraging techniques such as *just-in-time compilation* or *self-modifying code* [36, 58].

To compile C programs to the vnTinyRAM machine language, we ported the GCC compiler to this architecture, building on the work of [16].

See Figure 1 for a functionality comparison with prior circuit generators (for details, see [27, §2]).

Supported functionality	[66, 64, 56]	[16]	[27]	**this work**
side-effect free comp.	✓	✓	✓	✓
data-dep. mem. accesses	×	✓	✓	✓
data-dep. contr. flow	×	✓	×	✓
self-modifying code	×	×	×	✓
universality	×	×	×	✓

Figure 1: Functionality comparison among circuit generators.

(2) Our circuit generator efficiently handles *larger* arbitrary programs: the size of the generated circuit $C_{\ell,n,T}$ is $O\big((\ell+n+T)\cdot\log(\ell+n+T)\big)$ gates. Thus, the dependence on program size is *additive*, instead of multiplicative as in [16], where the generated (non-universal) circuit has size $\Theta\big((n+T)\cdot(\log(n+T)+\ell)\big)$. As Figure 2 shows, our efficiency improvement compared to [16] is not merely asymptotic but yields sizable concrete savings: as program size ℓ increases, our amortized per-cycle gate count is essentially unchanged, while that of [16] grows without bound, becoming orders of magnitudes more expensive.

$n = 10^2$	$\lvert C_{\ell,n,T}\rvert / T$		improvement
$T = 2^{20}$	[16]	**this work**	
$\ell = 10^3$	1,872	1,368	1.4×
$\ell = 10^4$	10,872	1,371	7.9×
$\ell = 10^5$	100,872	1,400	72.1×
$\ell = 10^6$	1,000,872	1,694	590.8×

Figure 2: Per-cycle gate count improvements over [16].

An efficiency comparison with other non-universal circuit generators [66, 64, 56, 27] is not well-defined. First, they support more restricted classes of programs, so a programmer must "write around" the limited functionality. Second, their efficiency is not easily specified, since the output circuit is ad hoc for the given program, and the only way to know its size is to actually run the circuit generator. We expect, and find, that such circuit generators perform better than ours for programs that are already "close to a circuit", and worse for programs rich in data-dependent memory accesses and control flow.

1.4.2 A new zk-SNARK for circuits

Our third contribution is a high-performance implementation of a zk-SNARK for arithmetic circuits.

(3) We improve upon and implement the protocol of Parno et al. [56]. Unlike previous zk-SNARK implementations [56, 16, 27], we do not use off-the-shelf cryptographic libraries. Rather, we create a tailored implementation of the requisite components: the underlying finite-field arithmetic, elliptic-curve group arithmetic, pairing-based checks, and so on.

To facilitate comparison with prior work, we instantiate our techniques for two specific algebraic setups: we provide an instantiation based on Edwards curves [33] at 80 bits of security (as in [16]), and an instantiation based on Barreto–Naehrig curves [9] at 128 bits of security (as in [56, 27]).

On our reference platform (a typical desktop), proof verification is fast: at 80-bit security, for an n-byte input to the circuit, verification takes $4.7 + 0.0004 \cdot n$ milliseconds, *regardless of circuit size*; at 128-bit security, it takes $4.8 + 0.0005 \cdot n$. The constant term dominates for small inputs, and corresponds to the verifier's pairing-based checks; in both cases, it is *less than half* the time for separately evaluating the 12 requisite pairings of the checks. We achieve this saving by merging parts of the pairings' computation in a protocol-dependent way — another reason for a custom implementation of the underlying math.

Key generation and proof generation entail a per-gate cost. For example, for a circuit with 16 million gates: at 80 bits of security, key generation takes 81 μs per gate and proving takes 109 μs per gate; at 128 bits of security, these per-gate costs mildly increase to 100 μs and 144 μs.

As in previous zk-SNARK implementations, proofs have constant size (independent of the circuit or input size); for us, they are 230 bytes at 80 bits of security, and 288 bytes at 128 bits of security.

Compared to previous implementations of zk-SNARKs for circuits [56, 16, 27], our implementation improves both proving and verification times, e.g., see Figure 3.

	80 bits of security			128 bits of security		
	[16]	**this**	impr.	[56]	**this**	impr.
Key gen.	306 s	97 s	3.2×	123 s	117 s	1.1×
Prover	351 s	115 s	3.1×	784 s	147 s	5.3×
Verifier	66.1 ms	4.9 ms	13.5×	9.2 ms	5.1 ms	1.8×
Proof	322 B	230 B	1.4×	288 B	288 B	(same)

Figure 3: Comparison with prior zk-SNARKs for a 1-million-gate arithmetic circuit and a 1000-bit input, running on our benchmarking machine, using software provided by the respective authors. Since [27] is a re-implementation of [56], we only include the latter's performance. ($N = 5$ and std $< 2\%$)

1.4.3 Two components: independent or combined

Our new circuit generator and our new zk-SNARK for circuits can be used independently. For instance, the circuit generator can (up to interface matching) replace the circuit generators in [66, 64, 56, 16, 27], thus granting these systems universality. Similarly, our zk-SNARK for circuits can replace the underlying zk-SNARKs in [56, 16, 27], or be used directly in applications where a suitable circuit is already specified.

Combining these two components, we obtain a full system: a *zk-SNARK for proving/verifying correctness of* vnTinyRAM *computations*; see Figure 4 and Figure 5 for diagrams of this system. We evaluated this overall system for programs with up to 10,000 instructions, running for up to 32,000 steps. Verification time is, again, only few milliseconds, independent of the running time of the vnTinyRAM program, even when program size and input size are kilobytes. Proofs, as mentioned, have a small constant size. Key generation and proof generation entail a per-cycle cost, with a dependence on program size that "tapers off" as computation length increases. For instance, at 128-bit security and vnTinyRAM with a word size of 32 bits, key generation takes 210 ms per cycle and proving takes 100 ms per cycle, for 8K-instruction programs.

JIT case study: efficient `memcpy`. Besides evaluating individual components, we give an example demonstrating the rich functionality supported by the integrated system. We wrote a vnTinyRAM implementation of `memcpy` that leverages *just-in-time compilation* (in particular, *dynamic loop unrolling*) to require fewer cycles. (See Section B.)

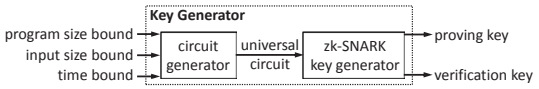


Figure 4: **Offline phase (once).** The key generator outputs a proving key and verification key, for proving and verifying correctness of any program execution meeting the given bounds.

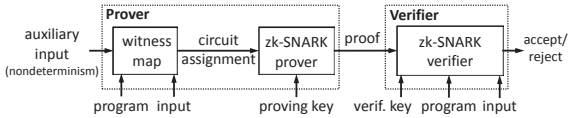


Figure 5: **Online phase (any number of times).** The prover sends a short and easy-to-verify proof to a verifier. This can be repeated any number of times, each time for a different program and input.

2 Preliminaries

$\mathbb{F}[z]$ denotes the ring of univariate polynomials over $\mathbb{F}$, and by $\mathbb{F}^{\leq d}[z]$ the subring of polynomials of degree $\leq d$. Concatenation of vectors/scalars is denoted by $\circ$.

2.1 Arithmetic circuits

Given a finite field $\mathbb{F}$, an $\mathbb{F}$-*arithmetic circuit* takes inputs that are elements in $\mathbb{F}$, and its gates output elements in $\mathbb{F}$. The circuits we consider only have *bilinear gates*.[1]

Definition 2.1. *Let n,h,l respectively denote the input, witness, and output size. The* **circuit satisfaction problem** *of a circuit $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ with bilinear gates is defined by the relation $\mathscr{R}_C = \{(\vec{x},\vec{a}) \in \mathbb{F}^n \times \mathbb{F}^h : C(\vec{x},\vec{a}) = 0^l\}$ and language $\mathscr{L}_C = \{\vec{x} \in \mathbb{F}^n : \exists \vec{a} \in \mathbb{F}^h,\ C(\vec{x},\vec{a}) = 0^l\}$.*

All the arithmetic circuits we consider are over prime fields $\mathbb{F}_p$. In this case, when passing boolean strings as inputs to arithmetic circuits, we *pack* the string's bits into as few field elements as possible: given $s \in \{0,1\}^m$, we use $[\![s]\!]_p^m$ to denote the vector $\vec{x} \in \mathbb{F}_p^{|m|_p}$, where $|m|_p := \lceil m / \lfloor \log p \rfloor \rceil$, such that the binary representation of $x_i \in \mathbb{F}_p$ is the i-th block of $\lceil \log p \rceil$ bits in s (padded with 0's if needed). We extend the notation $[\![s]\!]_p^m$ to binary strings $s \in \{0,1\}^n$ with $n < m$ bits via padding: $[\![s]\!]_p^m := [\![s0^{m-n}]\!]_p^m$.

2.2 Quadratic arithmetic programs

Our zk-SNARK leverages *quadratic arithmetic programs* (QAPs), introduced by Gennaro et al. [38].

Definition 2.2. *A* **quadratic arithmetic program** *of size m and degree d over $\mathbb{F}$ is a tuple $(\vec{A},\vec{B},\vec{C},Z)$, where $\vec{A},\vec{B},\vec{C}$ are three vectors, each of $m+1$ polynomials in $\mathbb{F}^{\leq d-1}[z]$, and $Z \in \mathbb{F}[z]$ has degree exactly d.*

[1] A gate with inputs $x_1,\ldots,x_m \in \mathbb{F}$ is *bilinear* if the output is $\langle \vec{a},(1,x_1,\ldots,x_m)\rangle \cdot \langle \vec{b},(1,x_1,\ldots,x_m)\rangle$ for some $\vec{a},\vec{b} \in \mathbb{F}^{m+1}$. In particular, these include addition, multiplication, and constant gates.

Like a circuit, a QAP induces a satisfaction problem:

Definition 2.3. *The* **satisfaction problem** *of a size-m QAP* $(\vec{A},\vec{B},\vec{C},Z)$ *is the relation* $\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$ *of pairs* $(\vec{x},\vec{s})$ *such that (i)* $\vec{x}\in\mathbb{F}^n$, $\vec{s}\in\mathbb{F}^m$, *and* $n\leq m$; *(ii)* $x_i=s_i$ *for* $i\in[n]$ *(i.e.,* $\vec{s}$ *extends* $\vec{x}$*); and (iii) the polynomial* $Z(z)$ *divides the following one:*

$$(A_0(z)+\textstyle\sum_{i=1}^m s_iA_i(z))\cdot(B_0(z)+\sum_{i=1}^m s_iB_i(z)) \\ -(C_0(z)+\textstyle\sum_{i=1}^m s_iC_i(z))\ .$$

We denote by $\mathscr{L}_{(\vec{A},\vec{B},\vec{C},Z)}$ *the language of* $\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$.

Gennaro et al. [38] showed that circuit satisfiability can be efficiently reduced to QAP satisfiability (which can then be proved and verified using zk-SNARKs):

Lemma 2.4. *There exist two polynomial-time algorithms* $\mathsf{QAPinst},\mathsf{QAPwit}$ *that work as follows. For any circuit* $C\colon\mathbb{F}^n\times\mathbb{F}^h\to\mathbb{F}^l$ *with a wires and b gates,* $(\vec{A},\vec{B},\vec{C},Z):=\mathsf{QAPinst}(C)$ *is a QAP of size m and degree d over* $\mathbb{F}$ *that satisfies the following three properties.*

- EFFICIENCY. *It holds that* $m=a$ *and* $d=b+l+1$.
- COMPLETENESS. *For any* $(\vec{x},\vec{a})\in\mathscr{R}_C$, *it holds that* $(\vec{x},\vec{s})\in\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, *where* $\vec{s}:=\mathsf{QAPwit}(C,\vec{x},\vec{a})$.
- PROOF OF KNOWLEDGE. *For any* $(\vec{x},\vec{s})\in\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, *it holds that* $(\vec{x},\vec{a})\in\mathscr{R}_C$, *where* $\vec{a}$ *is a prefix of* $\vec{s}$.
- NON-DEGENERACY. *The polynomials* $A_0,\ldots,A_n$ *are nonzero and distinct.*

2.3 Pairings

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic groups of order r. We denote elements of $\mathbb{G}_1,\mathbb{G}_2$ via calligraphic letters such as $\mathscr{P},\mathscr{Q}$. We write $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation. Let $\mathscr{P}_1$ be a generator of $\mathbb{G}_1$, i.e., $\mathbb{G}_1=\{\alpha\mathscr{P}_1\}_{\alpha\in\mathbb{F}_r}$ (α is also viewed as an integer, hence $\alpha\mathscr{P}_1$ is well-defined); let $\mathscr{P}_2$ be a generator for $\mathbb{G}_2$. A *pairing* is an efficient map $e\colon\mathbb{G}_1\times\mathbb{G}_2\to\mathbb{G}_T$, where $\mathbb{G}_T$ is also a cyclic group of order r (which we write in multiplicative notation), satisfying the following properties: (i) *bilinearity*: for every nonzero elements $\alpha,\beta\in\mathbb{F}_r$, it holds that $e(\alpha\mathscr{P}_1,\beta\mathscr{P}_2)=e(\mathscr{P}_1,\mathscr{P}_2)^{\alpha\beta}$; (ii) *non-degeneracy*: $e(\mathscr{P}_1,\mathscr{P}_2)$ is not the identity in $\mathbb{G}_T$.

2.4 zk-SNARKs for arithmetic circuits

A (preprocessing) **zk-SNARK** for $\mathbb{F}$-arithmetic circuit satisfiability (see, e.g., [22]) is a triple of polynomial-time algorithms (G,P,V), called *key generator*, *prover*, and *verifier*. The key generator G, given a security parameter λ and an $\mathbb{F}$-arithmetic circuit $C\colon\mathbb{F}^n\times\mathbb{F}^h\to\mathbb{F}^l$, samples a *proving key* pk and a *verification key* vk; these are the proof system's public parameters, which need to be generated only once per circuit. After that, anyone can use pk to generate non-interactive proofs for the language $\mathscr{L}_C$, and anyone can use the vk to check these proofs. Namely, given pk and any $(\vec{x},\vec{a})\in\mathscr{R}_C$, the honest prover $P(\mathsf{pk},\vec{x},\vec{a})$ produces a proof π attesting that $\vec{x}\in\mathscr{L}_C$; the verifier $V(\mathsf{vk},\vec{x},\pi)$ checks that π is a valid proof for $\vec{x}\in\mathscr{L}_C$. A proof π is both a proof of knowledge, and a (statistical) zero-knowledge proof. The succinctness property requires that π has length $O_\lambda(1)$ and V runs in time $O_\lambda(|\vec{x}|)$, where O_λ hides a (fixed) polynomial in λ.

Constructions. Several zk-SNARK constructions are known [45, 51, 38, 22, 56, 16, 52]. The most efficient ones are based on *quadratic span programs* (QSPs) [38, 52] or *quadratic arithmetic programs* (QAPs) [38, 22, 56, 16]. We focused on QAP-based constructions, because QAPs allow for tighter reductions from *arithmetic* circuits (see Lemma 2.4). Concretely, we build on the QAP-based zk-SNARK protocol of Parno et al. [56] (see Section 4).

Remark 2.5 (full succinctness)**.** The key generator G takes C as input, and so its complexity is linear in $|C|$. One could require G to *not* take C as input, and have its output keys work for *all* (polynomial-size) circuits C; then, G's running time would be independent of C. A zk-SNARK satisfying this stronger property is *fully succinct*. Theoretical constructions of such zk-SNARKs are known, based on various cryptographic assumptions [54, 69, 21]. Despite achieving essentially-optimal asymptotics [6, 18, 15, 14, 21] no implementations of them have been reported to date.

2.5 A von Neumann RISC architecture

Ben-Sasson et al. [16] introduced TinyRAM, a Harvard RISC architecture with word-addressable memory. We modify TinyRAM to obtain vnTinyRAM, which differs from it in two main ways. First, vnTinyRAM follows the *von Neumann paradigm*, whereby program and data are stored in the same read-write address space; programs may use runtime code generation. Second, vnTinyRAM has byte-addressable memory, along with instructions to load/store bytes or words.[2]

Besides the above main differences, vnTinyRAM is very similar to TinyRAM. Namely, it is parametrized by the *word size*, denoted W, and the *number of registers*, denoted K. The *CPU state* of the machine consists of (i) a W-bit *program counter*; (ii) K general-purpose W-bit *registers*; (iii) a 1-bit *condition flag*. The full state of the machine also includes *memory*, which is a linear array of 2^W bytes, and two *tapes*, each with a string of W-bit words, and read-only in one direction. One tape is for a *primary input* $\mathbb{x}$ and the other for an *auxiliary input* $\mathbb{w}$ (treated as nondeterministic, untrusted advice).

[2] Byte-addressing is common in programs performing array or string operations (and is a deeply-ingrained assumption in the GCC and LLVM compilers), while word-addressing in programs performing arithmetic.

In memory, an instruction is represented as a double word (one word for an immediate, and another for opcode, etc.). Thus, a *program* $\mathbb{P}$ is a list of address/double-word pairs specifying the initial contents of memory; all other memory locations assume the initial value of 0.

We define the language of accepting computations:

Definition 2.6. *Fix bounds ℓ, n, T. The language $\mathscr{L}_{\ell,n,T}$ consists of pairs $(\mathbb{P}, \mathbb{x})$ such that: (i) $\mathbb{P}$ is a program with $\leq \ell$ instructions, (ii) $\mathbb{x}$ is a primary input with $\leq n$ words, (iii) there exists an auxiliary input $\mathbb{w}$ s.t. $\mathbb{P}(\mathbb{x}, \mathbb{w})$ accepts in $\leq T$ steps. We denote by $\mathscr{R}_{\ell,n,T}$ the relation corresponding to $\mathscr{L}_{\ell,n,T}$.*

3 Our circuit generator

A circuit generator translates the correctness of suitably-bounded program executions into circuit satisfiability: given input bounds ℓ, n, T, it produces a circuit that can verify the execution of *any* program with $\leq \ell$ instructions, on *any* input of size $\leq n$, for $\leq T$ steps. More precisely, using the notations $[\![s]\!]_p$ (for packing the binary string s into field elements) and $|s|_p$ (for computing the number of field elements required to pack s) introduced in Section 2.1, we define a (universal) circuit generator for vnTinyRAM as follows.

Definition 3.1. *A* **(universal) circuit generator** *of efficiency $f(\cdot)$ over a prime field $\mathbb{F}_p$ is a polynomial-time algorithm* circ*, together with an efficient* witness map wit*, working as follows. For any program size bound ℓ, time bound T, and primary-input size bound n, $C := \mathsf{circ}(\ell, n, T)$ is an $\mathbb{F}_p$-arithmetic circuit $C\colon \mathbb{F}_p^m \times \mathbb{F}_p^h \to \mathbb{F}_p^l$, for $m := |\ell 2W|_p + |nW|_p$ and some h, l, where W is the word size (cf. Section 2.5).*

- EFFICIENCY. *The circuit C has $f(\ell, n, T)$ gates.*
- COMPLETENESS. *Given any program $\mathbb{P}$, primary input $\mathbb{x}$, and witness $\mathbb{w}$ such that $\big((\mathbb{P}, \mathbb{x}), \mathbb{w}\big) \in \mathscr{R}_{\ell,n,T}$, it holds that $(\vec{x}, \vec{a}) \in \mathscr{R}_C$, where $\vec{x} := [\![\mathbb{P}]\!]_p^{\ell 2W} \circ [\![\mathbb{x}]\!]_p^{nW}$ and $\vec{a} := \mathsf{wit}(\ell, n, T, \mathbb{P}, \mathbb{x}, \mathbb{w})$.*
- PROOF OF KNOWLEDGE. *There is a polynomial-time algorithm such that, given any $(\vec{x}, \vec{a}) \in \mathscr{R}_C$, outputs a witness $\mathbb{w}$ for $(\mathbb{P}, \mathbb{x}) \in \mathscr{L}_{\ell,n,T}$.*

The circuit C output by circ is *universal* because it does not depend on the program $\mathbb{P}$ or primary input $\mathbb{x}$, but only on their respective size bounds ℓ and n (as well as the time bound T). When combined with any proof system for circuit satisfiability (e.g., our zk-SNARK), this fact enables the generation of the proof systems' parameters to be universal as well. Namely, it is possible to generate keys for all bound choices (e.g., in powers of 2) up to some constant, *once and for all*; afterwards, one can pick the keys corresponding to bounds fitting a given computation. This avoids expensive per-program key generation and, more importantly, the need for a trusted party to conduct key generation anew for every program.

We construct a universal circuit generator with the following efficiency:

Theorem 3.2. *There is a circuit generator of efficiency $f(\ell, n, T) = O\big((\ell + n + T) \cdot \log(\ell + n + T)\big)$ over any prime field $\mathbb{F}_p$ of size $p > 2^{2W}$, where W is the word size (cf. Section 2.5).*

(In our case, the condition $p > 2^{2W}$ is always fulfilled.)

3.1 Past techniques

Most of the difficulties that arise when designing a circuit generator have to do with *data dependencies*. A circuit's topology does not depend on its inputs but, in contrast, program flow and memory accesses depend on the choice of program and the program's inputs. Thus, a circuit tasked with verifying program executions must be "ready" to support a multitude of program flows and memory accesses, despite the fact that its topology has already been fixed. Various techniques have been applied to the design of circuit generators.

Program analysis. In the extreme, if both the program $\mathbb{P}$ and its inputs $(\mathbb{x}, \mathbb{w})$ are known in advance, designing a circuit generator is simple: construct a circuit that evaluates $\mathbb{P}$ on $(\mathbb{x}, \mathbb{w})$ by preparing the circuit's topology to match the pre-determined program flow and memory accesses. But now suppose that only $\mathbb{P}$ is known in advance, but not its inputs $(\mathbb{x}, \mathbb{w})$. In this case, by analyzing $\mathbb{P}$ piece by piece (e.g., separately examine the various loops, branches, and so on), one could try to design a circuit $C_{\mathbb{P}}$ that can handle different choices of inputs. Most prior circuit generators [66, 64, 56, 27] take this approach.

However, this approach suffers from several limitations. First, the class of supported programs $\mathbb{P}$ is not rich, because support for data dependencies is limited. E.g., [56] requires array accesses and loop iteration bounds to be compile-time constants; also, while [27] supports data-dependent memory accesses, most program flow is also restricted to be known (or bounded) at compile-time; mitigations are possible, but only in special cases [72]. Second, and more importantly, this approach does not seem to allow for designing universal circuit generators, because the program $\mathbb{P}$ is *not known in advance* and thus there is no program to analyze.

Multiplex every access. Computers are universal random-access machines (RAMs), so one approach of designing a universal circuit is to mimic a computer's execution, building a layered circuit as follows. The i-th layer contains the entire state of the machine (CPU state and random-access memory) at time step i, and layer $i+1$ is computed from it by evaluating the transition function

of the machine, handling any accesses to memory via multiplexing. While this approach supports arbitrary program flow, memory accesses are inefficiently supported; indeed, if memory has S addresses, the resulting circuit is huge: it has size $\Omega(TS)$.

Nondeterministic routing. Ben-Sasson et al. [14] suggested using *nondeterministic routing* on a Beneš network to support memory accesses efficiently; Our circuit generator builds on the techniques of [14, 16], so we briefly review the main idea behind nondeterministic routing.

Following [14], Ben-Sasson et al. [16] introduced a simple computer architecture, called TinyRAM, and constructed a routing-based circuit generator for TinyRAM. They define the following notions. A *CPU state*, denoted S, is the CPU's contents (e.g., program counter, registers, flags) at a given time step. An *execution trace* for a program $\mathbb{P}$, time bound T, and primary input $\mathtt{x}$ is a sequence $\mathsf{tr} = (S_1, \ldots, S_T)$ of CPU states. An execution trace tr is *valid* if there is an auxiliary input $\mathtt{w}$ such that the execution trace induced by $\mathbb{P}$ running on inputs $(\mathtt{x}, \mathtt{w})$ is tr.

We seek an arithmetic circuit C for verifying that tr is valid. We break this down by splitting validity into three sub-properties: (i) *validity of instruction fetch* (for each time step, the correct instruction is fetched); (ii) *validity of instruction execution* (for each time step, the fetched instruction is correctly executed); and (iii) *validity of memory accesses* (each load from an address retrieves the value of the last store to that address).

The first two properties are verified as follows. Construct a circuit $C_{\mathbb{P}}$ so that, for any two CPU states S and S', $C_{\mathbb{P}}(S, S', g)$ is satisfied for some "guess" g if and only if S' can be reached from S (by fetching from $\mathbb{P}$ the instruction indicated by the program counter in S and then executing it), for *some* state of memory. Then, properties (i) and (ii) hold if $C_{\mathbb{P}}(S_i, S_{i+1}, \cdot)$ is satisfiable for $i = 1, \ldots, T-1$. Thus, C contains $T-1$ copies of $C_{\mathbb{P}}$, each wired to a pair of adjacent states in tr.

The third property is verified via nondeterministic routing. Assume that C also gets as input $\mathsf{MemSort}(\mathsf{tr})$, which equals to the sorting of tr by accessed memory addresses (breaking ties via timestamps), and write a circuit C_{mem} so that validity of memory accesses holds if C_{mem} is satisfied by each pair of adjacent states in $\mathsf{MemSort}(\mathsf{tr})$. (Roughly, C_{mem} checks consistency of "load-after-load", "load-after-store", and so on.) However, C merely gets some auxiliary input tr^*, which *purports* to be $\mathsf{MemSort}(\mathsf{tr})$. So C works as follows: (a) C has $T-1$ copies of C_{mem}, each wired to a pair of adjacent states in tr^*; (b) C separately verifies that $\mathsf{tr}^* = \mathsf{MemSort}(\mathsf{tr})$ by routing on a $O(T \log T)$-node Beneš network. The switches of the routing network are set according to non-deterministic guesses (i.e., additional values in the auxiliary input), and the routing network merely *verifies* that the switch settings induce a permutation; this allows for a very tight reduction. (Known constructions that *compute* the correct permutation hide large constants in big-oh notation [1].)

Past inefficiencies. After filling in additional details, the construction of [16] reviewed above gives a circuit of size $\Theta\big((n+T)\cdot(\log(n+T)+\ell)\big) = \Omega(\ell \cdot T)$. The $\Omega(\ell \cdot T)$ arises from the fact that all of the ℓ instructions in $\mathbb{P}$ are *hardcoded* into each of the $T-1$ copies of $C_{\mathbb{P}}$. Thus, besides being non-universal, the circuit scales inefficiently as ℓ grows (e.g., for $\ell = 10^4$, $C_{\mathbb{P}}$'s size is already dominated by $\mathbb{P}$'s size).

3.2 Our construction

In comparison to [16], our circuit generator is universal and, moreover, its size only grows with $\ell + T$ (additive dependence on program size) instead of with $\ell \cdot T$ (multiplicative dependence). As our evaluation demonstrates (see Section 5.1), the size improvement actually translates into significant savings in practice.

Instead of hardcoding the program $\mathbb{P}$ into each copy of the circuit $C_{\mathbb{P}}$, we follow the von Neumann paradigm, where the program $\mathbb{P}$ lies in the same read/write memory space as data. We ensure that $\mathbb{P}$ is loaded into the initial state of memory, using a dedicated circuit; we then verify instruction fetch via the *same* routing network that is used for checking data loads/stores. While the idea is intuitive, realizing it involves numerous technical difficulties, some of which are described below.

Routing instructions and data. We extend an execution trace to not only include CPU states but also instructions: $\mathsf{tr} = (S_1, I_1, \ldots, S_T, I_T)$ where S_i is the i-th CPU state, and I_i is the i-th executed instruction. We seek an arithmetic circuit C that checks tr, in this "extended" format, for the same three properties as above: (i) *validity of instruction fetch*; (ii) *validity of instruction execution*; (iii) *validity of memory accesses*.

As in [16], checking that tr satisfies property (ii) is quite straightforward. Construct a circuit C_{exe} so that, given two CPU states S, S' and an instruction I, $C_{\mathsf{exe}}(S, S', I, g)$ is satisfied, for some guess g, if and only if S' can be reached from S, by executing I, for some state of memory. Then, C contains $T-1$ copies of C_{exe}, each wired to adjacent CPU states and an instruction, i.e., the i-th copy is $C_{\mathsf{exe}}(S_i, S_{i+1}, I_i, g_i)$.

Unlike [16], though, we verify properties (i) and (iii) jointly, via the same routing network. The auxiliary input now contains $\mathsf{tr}^* = (A_1, \ldots, A_{2T})$, purportedly equal to the memory-sorted list of *both* instructions fetches and CPU states. (Since the program $\mathbb{P}$ lies in the same read-write memory as data, an instruction fetch from $\mathbb{P}$ is merely a special type of memory load.) Thus, to check that tr satisfies properties (i) and (iii), we design C to (a) verify that $\mathsf{tr}^* = \mathsf{MemSort}(\mathsf{tr})$ via nondeterministic routing, and

(b) verify validity of all (i.e., instruction and data) memory accesses, via a new circuit C'_{mem} applied to each pair of adjacent items A_i, A_{i+1} in $\mathtt{tr}^*$. Thus, in this approach, $\mathbb{P}$ is never replicated T times; rather, the fetching of its instructions is verified together with all other memory accesses, one instruction fetch at a time.

Multiple memory-access types. Each copy of C'_{mem} inspects a pair of items in $\mathtt{tr}^*$ and (assuming $\mathtt{tr}^* = \mathsf{MemSort}(\mathtt{tr})$) must ensure consistency of "load-after-load", "load-after-store", and so on. However, unlike in [16], the byte-addressable memory of vnTinyRAM is accessed in *different-sized* blocks: instruction-size blocks for instruction fetch; word-size blocks when loading/storing words; and byte-size blocks when loading/storing bytes. The consistency checks in C'_{mem} must handle "aliasing", i.e., accesses to the same point in memory via different addresses and block sizes.

We tackle this difficulty as follows. Double-word blocks are the largest blocks in which memory is accessed (as instructions are encoded as double words; cf. Section 2.5). We thus let each item in $\mathtt{tr}^*$ always specify a double-word, even if the item's memory access was with respect to a smaller-sized block (e.g., word or byte). With this modification, we can let C'_{mem} perform consistency checks "at the double-word level", and handling word/byte accesses by mapping them to double-word accesses with suitable shifting and masking.

Booting the machine. We have so far assumed that the program $\mathbb{P}$, given as input to C, *already* appears in memory. However, the circuit C sketched so far only verifies the validity of $\mathtt{tr}$ with respect to a machine whose memory is initialized to *some* state, corresponding to the execution of *some* program. But C must verify correct execution of, specifically, $\mathbb{P}$, and so it must also verify that memory is initialized to contain $\mathbb{P}$. Since C does not explicitly maintain memory (not even the initial one) and only implicitly reasons about memory via the routing network, it is not clear how C can perform this check.

We tackle this difficulty as follows. We further modify the the execution trace $\mathtt{tr}$, by extending it with an initial *boot* section, preceding the beginning of the computation, during which the input program $\mathbb{P}$ is stored into memory, one instruction $\mathbb{P}_i$ at a time. This extends the length of both $\mathtt{tr}$ and $\mathtt{tr}^*$ from $2T$ to $\ell + 2T$, for ℓ-instruction programs, and introduces a new type of item, "boot input store", in $\mathtt{tr}^*$. Similarly, the routing network is now responsible for routing $\ell + 2T$, rather than $2T$, packets.

Further optimizations. The above construction sketch (depicted in Figure 6) is only intuitive, and does not discuss other optimizations that ultimately yield the performance that we report in Section 5.1.

For example, while [16] rely on Beneš networks, we rely on *arbitrary-size Waksman networks* [10], which only require $N(\log N - 0.91)$ switches to route N packets, instead of $2^{\lceil \log N \rceil}(\lceil \log N \rceil - 0.5)$. Besides being closer to the information-theoretic lower bound of $N(\log N - 1.443)$, such networks eliminate costly rounding effects in [16], where the size of the network is *doubled* if N is just above a power of 2.

Compiling to vnTinyRAM. To enable verification of higher-level programs, written in C, we ported the GCC compiler to the vnTinyRAM architecture, by modifying the Harvard-architecture, word-addressible TinyRAM C compiler of [16]. Given a C program, written in the same subset of C as in [16], the compiler produces the initial memory map representing a program $\mathbb{P}$. This also served to validate the vnTinyRAM architectural choices (e.g., the move to byte-addressing significantly, and added instructions, improved efficiency for many programs).

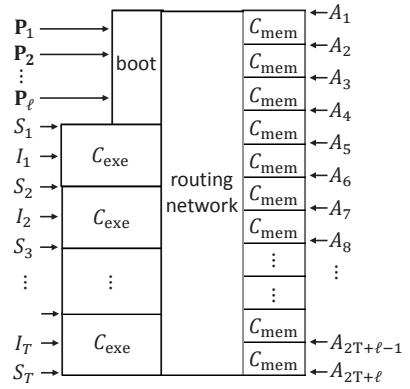


Figure 6: Outline of our universal circuit construction with the extended trace $\mathtt{tr}$ on the left and (allegedly) its memory sort $\mathtt{tr}^*$ on the right.

4 Our zk-SNARK for circuits

We discuss our second main contribution: a high-performance implementation of a zk-SNARK for arithmetic circuit satisfiability. Our approach is to *tailor* the requisite mathematical algorithms to the *specific* zk-SNARK protocol at hand. While our techniques can be instantiated in many algebraic setups and security levels, we demonstrate them in two specific settings, to facilitate comparison with prior work.

See Section 2.4 for an informal definition of a zk-SNARK for arithmetic circuit satisfiability. We improve upon and implement the zk-SNARK of Parno et al. [56]. For completeness the "PGHR protocol" is summarized in the full version of this paper, which provides pseudocode for its key generator G, prover P, and verifier V. The construction is based on QAPs, introduced in Section 2.2.

Like most other zk-SNARKs, the PGHR protocol relies on a *pairing*, which is specified by a prime $r \in \mathbb{N}$, three

cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order r, and a bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. (See Section 2.3.)

A pairing is typically instantiated via a *pairing-friendly elliptic curve*. Concretely, suppose that one uses a curve E defined over $\mathbb{F}_q$, with embedding degree k with respect to r, to instantiate the pairing. Then $\mathbb{G}_T$ is set to μ_r, the subgroup of r-th roots of unity in $\mathbb{F}^*_{q^k}$. The instantiation of $\mathbb{G}_1$ and $\mathbb{G}_2$ depends on the choice of e; typically, $\mathbb{G}_1$ is instantiated as an order-r subgroup of $E(\mathbb{F}_q)$, while, for efficiency reasons [7, 8], $\mathbb{G}_2$ as an order-r subgroup of $E'(\mathbb{F}_{k/d})$ where E' is a *d-th twist* of E. Finally, the pairing e is typically a two-stage function $e(\mathscr{P}, \mathscr{Q}) := \mathsf{FE}(\mathsf{ML}(\mathscr{P}, \mathscr{Q}))$, where $\mathsf{ML}\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_q^k$ is known as *Miller loop*, and $\mathsf{FE}\colon \mathbb{F}_q^k \to \mathbb{F}_q^k$ is known as *final exponentiation* and maps α to $\mathsf{FE}(\alpha) := \alpha^{(q^k-1)/r}$.

As mentioned, we instantiate our techniques based on two different curves: an Edwards curve for the 80-bit security level (as in [16]) and a Barreto–Naehrig curve for the 128-bits security level (as in [56, 27]). We selected both the Edwards curve and Barreto–Naehrig curve so that $r-1$ has high *2-adic order* (i.e., $r-1$ is divisible by a large power of 2), because this was shown to improve the efficiency of the key generator and the prover [16].

4.1 An optimized verifier

The verifier V takes as input a verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π, and checks if π is a valid proof for the statement "$\vec{x} \in \mathscr{L}_C$". The computation of V consists of two parts. First, use $\mathsf{vk}_{\mathsf{IC},0}, \ldots, \mathsf{vk}_{\mathsf{IC},n} \in \mathbb{G}_1$ (part of vk) and input $\vec{x}$ to compute $\mathsf{vk}_{\vec{x}} := \mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^{n} x_i \mathsf{vk}_{\mathsf{IC},i}$. Second, use vk, $\mathsf{vk}_{\vec{x}}$, and π, to compute 12 pairings and perform the required checks. In other words, V performs $O(n)$ scalar multiplications in $\mathbb{G}_1$, followed by $O(1)$ pairing evaluations.

With regard to V's first part, variable-base multi-scalar multiplication techniques can be used to reduce the number of $\mathbb{G}_1$ operations needed to compute $\mathsf{vk}_{\vec{x}}$ [16, 56]. With regard to V's second part, even if the pairing evaluations take constant time (independent of the input size n), these evaluations are very expensive and *dominate for small n*. Our focus here is to minimize the cost of these pairing evaluations.

When only making "black-box" use of a pairing, the verifier must evaluate 12 pairings, amounting to 12 Miller loops plus 12 final exponentiations. The straightforward approach is to compute these using a generic high-performance pairing library. We proceed differently: we obtain high-performance implementations of *subcomponents* of a pairing, and then tailor their use specifically to V's protocol.

Namely, first, we obtain state-of-the-art implementations of a Miller loop and final exponentiation. We utilize *optimal pairings* [70] to minimize the number of loop iterations in each Miller loop, and, to efficiently evaluate each Miller loop, rely on the formulas of [3] (for Edwards curves) and [20] (for BN curves). As for final exponentiation, we use multiple techniques to speed it up: [62, 44, 35, 50].

Next, building on the above foundation, we incorporate in V the following optimizations.

(1) Sharing Miller loops and final exponentiations. The verifier V computes two products of two pairings. We leverage the fact that a product of pairings can be evaluated faster than evaluating each pairing separately and then multiplying the results [60]. Concretely, in a product of m pairings, the Miller loop iterations for evaluating each factor can be carried out in "lock-step" so to share a single *Miller accumulator variable*, using one $\mathbb{F}_{q^k}$ squaring per loop instead of m.

In a similar vein, one can perform a single final exponentiation on the product of the outputs of the m Miller loops, instead of m final exponentiations and then multiplying the results. In fact, since the output of the pairing can be inverted for free (as the element is *unitary* so that inverting equals conjugating [61]), the idea of "sharing" final exponentiations extends to a ratio of pairing products. Thus, in the verifier we only need to perform 5, instead of 12, final exponentiations.

Our implementation incorporates both of the above techniques. For example, at the 80-bit security level, separately computing 12 optimal pairings costs 13.6 ms, but the above techniques reduce the time to only 8.1 ms. We decrease this further as discussed next.

(2) Precomputation by processing the verification key. Of the 12 pairings the verifier needs to evaluate, only one is such that both of its inputs come from the proof π. The other 11 pairings have one fixed input, either a generator of $\mathbb{G}_1$ or $\mathbb{G}_2$, or coming from the verification key vk.

When one input to a pairing is fixed, precomputation techniques apply [60], especially in the case when the fixed input is the *base point* in Miller's algorithm. In V, this holds for 9 out of the 11 pairing evaluations. We thus split the verifier's computation into an *offline phase*, which consists of a one-time precomputation that *only* depends on vk, and a many-time *online phase*, which depends on the precomputed values, input $\vec{x}$, and proof π. The result of the offline phase is a *processed verification key* vk^*. While vk^* is longer than vk, it allows the online phase to be faster.

E.g., at the 80-bit security level, vk^* decreases the total cost of pairing checks from 8.1 ms to 4.7 ms.

4.2 An optimized prover

The prover P takes as input a proving key pk (which includes the circuit $C\colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$), input $\vec{x} \in \mathbb{F}_r^n$, and witness $\vec{a} \in \mathbb{F}_r$. The prover P is tasked to produce a proof

π, attesting that $\vec{x} \in \mathscr{L}_C$. The computation of P consists of two main parts. First, compute the coefficients $\vec{h}$ of the polynomial $H(z) := \frac{A(z)B(z)-C(z)}{Z(z)}$, where $A,B,C \in \mathbb{F}_r[z]$ are derived from the QAP instance $(\vec{A},\vec{B},\vec{C},Z) := \mathsf{QAPinst}(C)$ and QAP witness $\vec{s} := \mathsf{QAPwit}(C,\vec{x},\vec{a})$. Second, use the coefficients $\vec{h}$, QAP witness $\vec{s}$, and public key pk to compute π.

With regard to the first part of P, the coefficients $\vec{h}$ can be efficiently computed via FFT techniques [16, 56]; our implementation follows [16], and leverages the high 2-adic order of $r-1$ for both of the elliptic curves we use. With regard to P's second part, computing π requires solving large instances of the following problem: given elements $\mathscr{Q}_1,\ldots,\mathscr{Q}_n$ all in $\mathbb{G}_1$ (or all in $\mathbb{G}_2$) and scalars $\alpha_1,\ldots,\alpha_n \in \mathbb{F}_r$, compute $\langle \vec{\alpha},\vec{\mathscr{Q}}\rangle := \alpha_1\mathscr{Q}_1 + \cdots + \alpha_n\mathscr{Q}_n$. Previous work [56, 16] has leveraged generic multi-scalar multiplication to compute π. We observe that these algorithms can be tailored to the specific scalar distributions arising in P. In P, the vector $\vec{\alpha}$ is one of two types: (i) $\vec{\alpha} \in \mathbb{F}_r^{d+1}$ and represents the coefficients of the degree-d polynomial H; or (ii) $\vec{\alpha} = (1 \circ \vec{s} \circ \delta_1 \circ \delta_2 \circ \delta_3) \in \mathbb{F}_r^{4+m}$, for random $\delta_1,\delta_2,\delta_3 \in \mathbb{F}_r$.

In case i, the entries in of $\vec{\alpha}$ are random-looking. We use the Bos–Coster algorithm [26] due to its lesser memory requirements (as compared to, e.g., [57]). We follow [19]'s suggestions and achieve an assembly-optimized heap to implement the Bos–Coster algorithm.

In case ii, the entries in $\vec{s}$ depend on the input $(C,\vec{x},\vec{a})$ to QAPwit; in turn, $(C,\vec{x},\vec{a})$ depends on our circuit generator (Section 3). Using the above algorithm "as is" is inefficient: the algorithm works well when all the scalars have roughly the same bit complexity, but the entries in $\vec{c}$ have very different bit complexity. Indeed, $\vec{\alpha}$ equals to $\vec{s}$ augmented with a few entries; and $\vec{s}$, the QAP witness, can be thought of as the list of wire values in C when computing on $(\vec{x},\vec{a})$; the bit complexity of a wire value depends on whether it is storing a boolean value, a word value, and so on. We observe that there are only a few "types" of values, so that the entries of $\vec{\alpha}$ can be clustered into few groups of scalars with approximately the same bit complexity; we then apply the algorithm of [26] to each such group.

4.3 An optimized key generator

The key generator G takes as input a circuit $C\colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$, and is tasked to compute a proving key pk and a verification key vk. The computation of G consists of two main parts. First, evaluate each A_i,B_i,C_i at a random element τ, where $(\vec{A},\vec{B},\vec{C},Z) := \mathsf{QAPinst}(C)$ is the QAP instance. Second, use these evaluations to compute pk and vk.

With regard to G's first part, we follow [16] and again leverage the fact that $\mathbb{F}_r$ has a primitive root of unity of large order. With regard to G's second part, it is dominated by the cost of computing pk, which requires solving large instances of the following problem: given an element $\mathscr{P}$ in $\mathbb{G}_1$ or $\mathbb{G}_2$ and scalars $\alpha_1,\ldots,\alpha_n \in \mathbb{F}_r$, compute $\alpha_1\mathscr{P},\ldots,\alpha_n\mathscr{P}$. Previous work [56, 16], used fixed-base windowing [28] to efficiently compute such fixed-base multi-scalar multiplications.

In our implementation, we achieve additional efficiency, in space rather than in time. Specifically, we leverage a structural property of QAPs derived from arithmetic circuits, in order to reduce the size of the proving key pk, as we now explain. Lemma 2.4 states that an $\mathbb{F}$-arithmetic circuit $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$, with α wires and β gates, can be converted into a corresponding QAP of size $m = \alpha$ and degree $d \approx \beta$ over $\mathbb{F}$. Roughly, this is achieved in two steps. First, construct three matrices $\mathbf{A},\mathbf{B},\mathbf{C} \in \mathbb{F}^{(m+1)\times d}$ that encode C's topology: for each $j \in [d]$, the j-th column of $\mathbf{A},\mathbf{B}$ respectively encodes the "left" and "right" coefficients of the j-th bilinear gate in C, while the j-th column of $\mathbf{C}$ encodes the coefficients of the gate's output. Second, letting $S \subset \mathbb{F}$ be a set of size d, define $Z(z) := \prod_{\omega\in S}(z-\omega)$ and, for $i \in \{0,\ldots,m\}$, let A_i be the low-degree extension of the i-th row of $\mathbf{A}$; similarly define each B_i and C_i. All prior QAP-based zk-SNARK implementations exploit the fact that *columns* in the matrices $\mathbf{A},\mathbf{B},\mathbf{C}$ are very sparse.

In contrast, we also leverage a *different* kind of sparsity: we observe that it is common for *entire rows* of $\mathbf{A},\mathbf{B},\mathbf{C}$ to be all zeroes, causing the corresponding low-degree extensions to be zero polynomials.[3] For instance, our circuit generator typically outputs a circuit for which the percentage of non-zero polynomials in $\vec{A},\vec{B},\vec{C}$ is only about $52\%, 15\%, 71\%$ respectively. The fact that many polynomials in $\vec{A},\vec{B},\vec{C}$ evaluate to zero can be used towards reducing the size of pk, by switching from a dense representation to a sparse one.

In fact, we have *engineered* our circuit generator to reduce the number of non-zero polynomials in $\vec{B}$ as much as possible, because computations associated to evaluations of $\vec{B}$ are conducted with respect to more expensive $\mathbb{G}_2$ arithmetic, which we want to avoid as much as possible.

5 Evaluation

We evaluated our system on a desktop computer with a 3.40 GHz Intel Core i7-4770 CPU (with Turbo Boost disabled) and 32 GB of RAM. All experiments, except the largest in Figure 8 and 9, used a small fraction of the RAM. For the two largest experiments in Figure 9 we added a Crucial M4 solid state disk for swap space. (While our code supports multi-threading, our experiments are in single-thread mode, for comparison with prior work.)

[3] E.g., if the i-th wire never appears with a non-zero coefficient as the "left" input of a bilinear gate, then the i-th row of $\mathbf{A}$ is zero.

5.1 Performance of our circuit generator

In Section 3 we described our universal circuit generator; we now benchmark its performance.

Parameters. The circuit supports vnTinyRAM, which is parametrized by two quantities: the *word size* W and the *number of registers* K (see Section 2.5). We report performance for a machine with $K = 16$ registers, and two choices of word size: $W = 16$ and $W = 32$.

Methodology. Theorem 3.2 provides an asymptotic efficiency guarantee: it states that our circuit generator has efficiency $f(\ell,n,T) = O\big((\ell+n+T)\cdot\log(\ell+n+T)\big)$. To understand concrete efficiency, we "uncover" the constants hidden in the *big-oh* notation. By studying the number of gates in various subcircuits of the generated circuit $C := \mathsf{circ}(\ell,n,T)$, we computed the following (quite tight) upper bound on C's size:

$$(12+2W)\cdot\ell+(12+W)\cdot n+|C_{\mathsf{exe}}|\cdot T+(|C_{\mathsf{mem}}|+ 4\log H-1.82)\cdot H$$

where $H := (\ell+n+2T)$ is the "height" of the routing network, and

- for $(W,K) = (16,16)$: $|C_{\mathsf{exe}}| = 777$ and $|C_{\mathsf{mem}}| = 211$;
- for $(W,K) = (32,16)$: $|C_{\mathsf{exe}}| = 1114$ and $|C_{\mathsf{mem}}| = 355$.

In Figure 7, we give per-cycle gate counts (i.e., $|C|/|T|$) for various choices of (ℓ,n,T); we also give sub-counts divided among program/input boot, CPU execution, memory checking, and routing. (See the full version of this paper for an extended table with additional data.)

Discussion. We first go through the size expression, to understand it: The first two terms, $(12+2W)\cdot\ell+(12+W)\cdot n$, correspond to the pre-execution boot phase, during which an ℓ-instruction program and an n-word primary input are loaded into the machine. The term $|C_{\mathsf{exe}}|\cdot T$ corresponds to the T copies of C_{exe} used to verify each CPU transition, given the fetched instruction and two CPU states. The term $|C_{\mathsf{mem}}|\cdot H$ corresponds to the H copies of C_{mem} used to verify consistency on the memory-sorted trace. Finally, the term $(4\log H-1.82)\cdot H$ corresponds to the routing network for routing H packets (two gates for each of $(2\log H-0.91)\cdot H$ binary switches). Note that $H = (\ell+n+2T)$ because boot needs $\ell+n$ memory stores (one for each program instruction and primary input word) and execution needs $2T$ memory accesses (1 instruction fetch and 1 data store/load per execution cycle).

The gate counts in Figure 7 demonstrate the additive (instead of multiplicative) dependence on program size of our universal circuit pays off. For example, for $(W,K) = (32,16)$, a 100-fold increase in program size, from $\ell = 10^3$ to $\ell = 10^5$, barely impacts the per-cycle gate count: for $T = 2^{20}$, it increases from 1,992.5 to only 2,041.5. Indeed, the cost of program size is incurred, once and for all, during the machine boot; Figure 7 shows that the per-cycle cost of machine boot diminishes as T grows.

Second, *less than half* of C's gates are dedicated to verifying accesses to random-access memory, while the majority of gates are dedicated to verifying execution of the CPU; indeed, almost always, $|C_{\mathsf{exe}}|T > \frac{1}{2}|C|$. Put otherwise, C, which verifies an automaton with random-access memory (vnTinyRAM), has size that is less than twice that for verifying an automaton with the same CPU but no random-access memory at all. Moreover, note that the size of C_{exe} appears quite tight: for example, with $(W,K) = (32,16)$, it has size 1114, not much larger than the size of the CPU state (545 bits).

5.2 Performance of our zk-SNARK for circuit satisfiability

In Section 4 we described our zk-SNARK implementation; we now benchmark its performance.

Methodology. We provide performance characteristics for each of the zk-SNARK algorithms, G, P and V, at the 80-bit and 128-bit security levels.

(1) The key generator G takes as input an arithmetic circuit C: $\mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$. Its efficiency mostly depends on the number of gates and wires in C, because these affect the size and degree of the corresponding QAP (see Lemma 2.4). Thus, we evaluate G on a circuit with 2^i gates and 2^i wires for $i \in \{10,12,\ldots,24\}$ (and fixed $n = h = l = 100$). In Figure 8 we report the resulting running times and key sizes, as *per-gate costs*.

(2) The prover P takes as input a proving key pk, input $\vec{x} \in \mathbb{F}_r^n$, and witness $\vec{a} \in \mathbb{F}_r^h$. Its efficiency mostly depends on the number of gates and wires in C (the circuit used to generate pk); we thus evaluate P on the proving keys output by G, for the same circuits as above. In Figure 8 we report the times, as per-gate costs, and proof sizes.

(3) The verifier V takes as input a verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π. Its efficiency depends only on $\vec{x}$ (since the size of $\vec{x}$ determines that of vk). Thus, we evaluate V on a random input $\vec{x} \in \mathbb{F}_r^n$ of 2^i bytes for $i \in \{2,4,\ldots,20\}$. In Figure 8 we report the resulting running times, along with corresponding key sizes.

Discussion. The data demonstrates that our zk-SNARK implementation works and scales as expected, as long as sufficient memory is available (e.g., on a desktop computer with 32GB of DRAM: up to 16 million gates). Key generation takes about 10 ms per gate of C; the size of a proving key is about 300 B per gate, and the size of a verification key is about 1 B per byte of input to C. Running the prover takes 11 ms to 14 ms per gate. For an n-byte input, proof verification time is $c_1 n + c_0$, where c_0 is a few milliseconds and c_1 is a few tenths of microseconds.

5.3 Performance of the combined system

As discussed, our circuit generator (Section 3) and zk-SNARK for circuits (Section 4) can be used in-

		Per-cycle gate count of $C := \mathsf{circ}(\ell,n,T)$ with vnTinyRAM parameters (W,K)									
		$n = 10^2, K = 16$									
		$W = 16$					$W = 32$				
		$\lvert C\rvert/T$	$\lvert C\rvert/T$ divided among				Per cycle	$\lvert C\rvert/T$ divided among			
			boot	exec.	mem.	routing		boot	exec.	mem.	routing
$\ell = 10^3$	$T = 2^{20}$	1,367.4	0.04	777.0	422.2	168.1	1,992.5	0.08	1,114.0	710.4	168.1
	$T = 2^{24}$	1,399.0	0.00	777.0	422.0	200.0	2,024.0	0.00	1,114.0	710.0	200.0
	$T = 2^{28}$	1,431.0	0.00	777.0	422.0	232.0	2,056.0	0.00	1,114.0	710.0	232.0
$\ell = 10^4$	$T = 2^{20}$	1,370.3	0.41	777.0	424.0	168.8	1,997.0	0.72	1,114.0	713.4	168.8
	$T = 2^{24}$	1,399.2	0.03	777.0	422.1	200.1	2,024.3	0.05	1,114.0	710.2	200.1
	$T = 2^{28}$	1,431.0	0.00	777.0	422.0	232.0	2,056.0	0.00	1,114.0	710.0	232.0
$\ell = 10^5$	$T = 2^{20}$	1,399.7	4.12	777.0	442.1	176.4	2,041.5	7.19	1,114.0	743.9	176.4
	$T = 2^{24}$	1,401.1	0.26	777.0	423.3	200.6	2,027.2	0.45	1,114.0	712.1	200.6
	$T = 2^{28}$	1,431.1	0.02	777.0	422.1	232.0	2,056.2	0.03	1,114.0	710.1	232.0

Figure 7: Per-cycle gate counts in $C := \mathsf{circ}(\ell,n,T)$ for different choices of (ℓ,n,T) and vnTinyRAM parameters (W,K).

		80 bits of security		128 bits of security	
key gen. G		time/$\lvert C\rvert$	$\lvert\mathsf{pk}\rvert/\lvert C\rvert$	time/$\lvert C\rvert$	$\lvert\mathsf{pk}\rvert/\lvert C\rvert$
$n = 100$	$\lvert C\rvert = 2^{10}$	0.21 ms	248.8 B	0.21 ms	304.1 B
	$\lvert C\rvert = 2^{12}$	0.16 ms	252.5 B	0.17 ms	309.1 B
	$\lvert C\rvert = 2^{14}$	0.14 ms	253.4 B	0.16 ms	310.3 B
	$\lvert C\rvert = 2^{16}$	0.12 ms	253.7 B	0.14 ms	310.6 B
	$\lvert C\rvert = 2^{18}$	0.11 ms	253.7 B	0.12 ms	310.7 B
	$\lvert C\rvert = 2^{20}$	0.10 ms	253.7 B	0.12 ms	310.7 B
	$\lvert C\rvert = 2^{22}$	0.09 ms	253.7 B	0.11 ms	310.7 B
	$\lvert C\rvert = 2^{24}$	0.08 ms	253.7 B	0.10 ms	310.7 B
	$\lvert\mathsf{vk}\rvert$	2.8 KB		3.6 KB	
prover P		time/$\lvert C\rvert$	$\lvert\pi\rvert$	time/$\lvert C\rvert$	$\lvert\pi\rvert$
$n = 100$	$\lvert C\rvert = 2^{10}$	0.18 ms	230 B	0.21 ms	288 B
	$\lvert C\rvert = 2^{12}$	0.16 ms	230 B	0.18 ms	288 B
	$\lvert C\rvert = 2^{14}$	0.14 ms	230 B	0.16 ms	288 B
	$\lvert C\rvert = 2^{16}$	0.13 ms	230 B	0.15 ms	288 B
	$\lvert C\rvert = 2^{18}$	0.12 ms	230 B	0.15 ms	288 B
	$\lvert C\rvert = 2^{20}$	0.12 ms	230 B	0.15 ms	288 B
	$\lvert C\rvert = 2^{22}$	0.11 ms	230 B	0.14 ms	288 B
	$\lvert C\rvert = 2^{24}$	0.11 ms	230 B	0.14 ms	288 B
verifier V		$\lvert\mathsf{vk}\rvert/\lvert\vec{x}\rvert$	time/$\lvert\vec{x}\rvert$	$\lvert\mathsf{vk}\rvert/\lvert\vec{x}\rvert$	time/$\lvert\vec{x}\rvert$
$\lvert\vec{x}\rvert = 4$ B		118.7 B	1.2 ms	123.4 B	1.2 ms
$\lvert\vec{x}\rvert = 16$ B		29.7 B	0.3 ms	30.8 B	0.3 ms
$\lvert\vec{x}\rvert = 64$ B		8.1 B	76.7 μs	8.7 B	81.2 μs
$\lvert\vec{x}\rvert = 256$ B		2.8 B	19.5 μs	2.9 B	20.3 μs
$\lvert\vec{x}\rvert = 1.0$ KB		1.5 B	5.4 μs	1.5 B	5.9 μs
$\lvert\vec{x}\rvert = 4.1$ KB		1.1 B	1.8 μs	1.1 B	2.1 μs
$\lvert\vec{x}\rvert = 16.4$ KB		1.1 B	0.8 μs	1.0 B	1.0 μs
$\lvert\vec{x}\rvert = 65.5$ KB		1.0 B	0.5 μs	1.0 B	0.7 μs
$\lvert\vec{x}\rvert = 262.1$ KB		1.0 B	0.4 μs	1.0 B	0.6 μs
$\lvert\vec{x}\rvert = 1.0$ MB		1.0 B	0.4 μs	1.0 B	0.5 μs

Figure 8: Per-gate costs of the key generator and prover; and per-byte costs of the verifier. ($N = 10$ and std $< 1\%$)

dependently, or combined to obtain a zk-SNARK for vnTinyRAM. For completeness, the paper's full version we spell out how these two components can be combined. Here we report measured performance of this combined system, at the 128-bit security level, and for a word size $W = 32$ and number of registers $K = 16$.

Methodology. A zk-SNARK for vnTinyRAM is a triple of algorithms $(\mathsf{KeyGen},\mathsf{Prove},\mathsf{Verify})$. Given bounds ℓ,n,T (for program size, input size, and time), the efficiency of KeyGen and Prove depends on ℓ,n,T, while that of Verify essentially depends only on ℓ,n. Thus, we benchmark the system as follows. We evaluate KeyGen and Prove for various choices of ℓ and T, while keeping $n = 100$. Instead, since the efficiency of Verify does not depend on T, we evaluate Verify, for various choices of ℓ and n, on random ℓ-instruction programs and n-word inputs. In Figure 9, we report the following measurements: KeyGen's running time, the sizes of the keys pk and vk, Prove's runtime, the (constant) proof size, and Verify's running time. For quantities growing with T, we divide by T and report the per-cycle cost.

Discussion. The measurements demonstrate that, on a desktop computer, our zk-SNARK for vnTinyRAM scales up to computations of 32,000 machine cycles, for programs with up to 10,000 instructions. Key generation takes about 200 ms per cycle; the size of a proving key is 500 KB to 650 KB per cycle, and the size of a verification key is a few kilobytes. Running the prover takes 100 ms to 200 ms per cycle. Verification times remain a few ms , even for inputs and programs of several kilobytes.

Program-specific vk**.** The time complexity of Verify is $O(\ell+n)$, so verification time grows with program size. This is inevitable, because Verify must *read* a program $\mathbb{P}$ (of at most ℓ instructions) and input $\mathbb{x}$ (of at most n words) in order to check, via the given proof π, if $(\mathbb{P},\mathbb{x}) \in \mathscr{L}_{\ell,n,T}$ (cf. Definition 2.6). However, this is inconvenient, e.g., when one has to verify many proofs relative to different inputs to the *same* program $\mathbb{P}$. In our zk-SNARK it is possible to amortize this cost as follows. Given vk and $\mathbb{P}$, one can derive, in time $O(\ell)$, a *program-specific* verification key $\mathsf{vk}_{\mathbb{P}}$, which can be used to verify proofs relative to any input to $\mathbb{P}$. Subsequently, the time complexity of Verify for any input $\mathbb{x}$ (to $\mathbb{P}$) is $O(n)$, *independent* of ℓ.

5.4 Comparison with prior work

5.4.1 Comparison with prior circuit generators

Universality is the main innovative feature of our circuit generator. *No* previous circuit generator achieves univer-

				128 bits of security				
				$W = 32, K = 16$				
				$\ell = 2$K	$\ell = 4$K	$\ell = 6$K	$\ell = 8$K	$\ell = 10$K
KeyGen	time/T	$n = 100$	$T = 4$K	209.8 ms	232.1 ms	257.5 ms	275.9 ms	306.4 ms
			$T = 8$K	190.9 ms	205.9 ms	216.1 ms	228.9 ms	238.8 ms
			$T = 16$K	195.4 ms	198.1 ms	204.2 ms	213.6 ms	218.3 ms
			$T = 32$K	206.0 ms	208.4 ms	211.2 ms	213.5 ms	223.7 ms
	|pk|/T		$T = 4$K	584.2 KB	653.6 KB	727.1 KB	784.0 KB	876.8 KB
			$T = 8$K	552.4 KB	585.2 KB	618.1 KB	655.1 KB	683.7 KB
			$T = 16$K	539.4 KB	553.9 KB	570.4 KB	586.9 KB	605.5 KB
			$T = 32$K	533.8 KB	541.1 KB	548.3 KB	555.6 KB	563.4 KB
	|vk|		$T = *$	17.0 KB	33.1 KB	49.2 KB	65.3 KB	81.5 KB
Prove	time/T	$n = 100$	$T = 4$K	75.7 ms	86.7 ms	103.4 ms	104.8 ms	133.7 ms
			$T = 8$K	69.2 ms	79.7 ms	97.0 ms	110.4 ms	113.0 ms
			$T = 16$K	89.0 ms	89.1 ms	98.4 ms	99.6 ms	103.3 ms
			$T = 32$K	98.9 ms	98.6 ms	102.3 ms	102.1 ms	114.2 ms
Verify	time	(indep. of T)	$n = 0$	19.0 ms	30.0 ms	40.6 ms	51.2 ms	61.3 ms
			$n = 10$	19.1 ms	30.2 ms	40.7 ms	51.2 ms	61.4 ms
			$n = 10^2$	19.6 ms	30.7 ms	41.3 ms	51.8 ms	61.9 ms
			$n = 10^3$	23.0 ms	34.1 ms	44.7 ms	55.2 ms	65.4 ms
			$n = 10^4$	48.9 ms	60.0 ms	70.6 ms	81.1 ms	91.3 ms

Figure 9: Per-cycle costs of KeyGen and Prove for various program sizes ℓ, and total running time of Verify for various ℓ and n.

sality. (See Figure 1 and Section 3.)

Putting universality aside and focusing on efficiency instead, a comparison with previous circuit generators is a multi-faceted problem. On one hand, due to a shared core of techniques, a comparison with [16]'s circuit generator is straightforward, and shows significant improvements in circuit size, especially as program size grows. See Section 1.4.1 and Figure 2 (the figure is for $W, K = 16$).

Instead, a comparison with other circuit generators [66, 64, 56, 27] is complex. First, they support a smaller class of programs (see Figure 1), so a programmer must "write around" the limited functionality, somehow. And second, their efficiency is not easily specified: due to the program-analysis techniques (see Section 3.1) the output circuit is ad hoc for the given program, and the only way to know its size is to actually run the circuit generator.

Compared to [66, 64, 56, 27], our circuit generator performs better for programs that are rich in memory accesses and control flow, and worse for programs that are more "circuit like".

Comparison with [66, 64, 56]. The circuit generators in [66, 64, 56] restrict loop iteration bounds and memory accesses to be known at compile time; if a program does not respect these restrictions, it must be first somehow mapped to another one that does. For simplicity, we take [56]'s circuit generator (the latest one) as representative and, to illustrate the differences between [56]'s and our circuit generator, we consider two "extremes".

On one extreme, we wrote a simple C program multiplying two 10×10 matrices of 16-bit integers. The circuit generator in [56] produces a circuit with 1100 gates; instead, our circuit generator (when given the corresponding vnTinyRAM assembly) produces a much larger circuit: one with $\approx 10^7$ gates.

On the other extreme, we consider a program making many random accesses to memory: pointer-chasing. Given a permutation π of $[N]$, start position $i \in [N]$, and an integer k, the program outputs $\pi^k(i)$, the element obtained by starting from i and following "pointers" for k times. Since no information about π is known at compile time, the only way of obtaining $\pi(j)$, the pointer to follow, in [56] is via a *linear scan*. On a simple C program that does one linear scan of π to obtain each new pointer, [56]'s generator outputs a circuit with $2Nk+1$ gates (each of the k array accesses costs $2N$ gates).

In vnTinyRAM, the corresponding program $\mathbb{P}$ consists of 9 instructions, and the input $\mathtt{x}$ to it is $N+3$ words. Booting vnTinyRAM with $\mathbb{P}$ and $\mathtt{x}$ requires $9+N+3$ "boot stores" (see Section 3.2), and takes $5+4k$ cycles to execute (independent of N). Say that we fix $k = 10$; then, in our circuit generator (with $W = 32$ and $K = 16$), each cycle costs about 2000 gates, and can perform a random access to memory. Thus, pointer chasing in our case is cheaper than in [56] already for $N > 5000$, and the multiplicative saving, which is about $\frac{20N}{2000 \cdot (5+40)} = \frac{N}{4500}$, grows unbounded as N increases.

Comparison with [27]. The circuit generator of [27] is also based on program analysis, but provides an additional feature that allows data-dependent memory accesses: a program may access memory by guessing the value and verifying its validity via a subcircuit that checks Merkle-tree authentication paths. In [27], memory consists of 2^{30} cells, and each access costs many gates: 140K for a load, and 280K for a store. In comparison, in our circuit generator for vnTinyRAM (with word size $W = 32$ so that memory has 2^{32} cells), each memory store/load costs less than 1000 gates out of about 2000 per cycle (see Section 5.1). Besides the aforementioned feature, [27] rely on program analysis, and (as in [66, 64, 56]) only support bounded control flow. Thus, [27] performs better than our circuit generator for programs with bounded control flow and few data-dependent accesses to memory.

5.4.2 Comparison with prior zk-SNARKs

Addressing the other component of our system, the zk-SNARK for circuits: Figure 3 compares our implementation with prior ones, on a 1-million-gate circuit with a 1000-byte input. As shown, we mildly improve the key generation time and, more importantly, significantly improve the "online" costs of proving and verification.

6 Conclusion

We have presented two main contributions: (i) a circuit generator for a von Neumann RISC architecture that is *universal* and scales *additively* with program size; and (ii) a high-performance zk-SNARK for arithmetic circuit satisfiability. These two components can be used independently to the benefit of other systems, or combined into a zk-SNARK that can prove/verify correctness of computations on this architecture.

The benefits of universality. Universality attains the conceptual advance of *once-and-for-all key generation*, allowing verifying all programs up to a given size. This removes major issues in prior systems: expensive per-program key generation and the thorny issue of conducting it anew in a trusted way for every program.

The price of universality. The price of universality is still very high. Going forward, and aiming for widespread use in security applications, more work is required to slash costs of key generation and proving so to scale up to larger computations: e.g., billion-gate circuits, or millions of vnTinyRAM cycles, and beyond. An interesting open problem is whether the "program analysis" techniques underlying most prior circuit generators [66, 64, 56, 27], typically more efficient for restricted classes of programs, can be used to construct universal circuits.

Beyond vnTinyRAM. Finally, going beyond the foundation of a von Neumann RISC architecture, more work lies ahead towards a richer architecture (e.g., efficient support for floating-point arithmetic and cryptographic acceleration), code libraries, and tighter compilers.

A Other prior work

Prior work most relevant to us is about zk-SNARKs, and is discussed in Section 1.2. There are also numerous works studying variations or relaxations of the goal we consider; here, we summarize some of them.

Interactive proofs for low-depth circuits. Goldwasser et al. [42] obtained an interactive proof for outsourcing computations of *low-depth circuits*. A set of works [32, 68, 67] has optimized and implemented the protocol of [42]. The protocol of [42] can also be reduced to a two-message argument system [48, 47]. Canetti et al. [30] showed how to extend the techniques in [42] to also handle non-uniform circuits.

Batching arguments. Ishai et al. [46] constructed a *batching argument* for NP, where, to simultaneously verify that N circuits of size S are satisfiable, the verifier runs in time $\max\{S^2, N\}$.

A set of works [63, 65, 66, 64] has improved, optimized, and implemented the batching argument of Ishai et al. [46] for the purpose of outsourcing computation. In particular, by relying on quadratic arithmetic programs of [38], Setty et al. [64] have improved the running time of the verifier and prover to $\max\{S,N\} \cdot \text{poly}(\lambda)$ and $\tilde{O}(S) \cdot \text{poly}(\lambda)$ respectively. Vu et al. [71] provide a system that incorporates both the batching arguments of [63, 65, 66, 64] as well as the interactive proofs of [32, 68, 67]. The system decides which of the two approaches is more efficient to use for outsourcing a given computation.

Braun et al. [27] apply batching techniques (as well as zk-SNARKs) to verify MapReduce computations, by relying on various verifiable data structures.

Arguments with competing provers. Canetti et al. [29] use collision-resistant hashes to get a protocol for outsourcing deterministic computations in a model where a verifier interacts with two computationally-bounded provers at least one of which is honest [34]. The protocol in [29] works *directly* for random-access machines, and therefore does not require reducing random-access machines to any "lower-level" representation (such as circuits). Canetti et al. implement their protocol for deterministic x86 programs.

Previous circuit generators. Some prior work addresses the problem of translating high-level languages into low-level languages such as circuits. Most prior work only supports restricted classes of programs: [66, 64] present a circuit generator based on Fairplay [53, 12], whose SFDL language does not support important primitives and has inefficient support for others; [56] present a circuit generator for programs without data dependencies (pointers and array indices must be known at compile time, and so do loop iteration bounds).

Other works support more general functionality: [16] rely on nondeterministic routing to support random-access machine computations [14]; [27] rely on online memory checking [24, 14] to support accessing untrusted storage from a circuit. See [27, Section 2] for a more detailed overview of some of the above techniques.

Other cryptographic tools. *Fully-homomorphic encryption* (FHE) [39] and *probabilistically-checkable proofs* [5, 4] are powerful tools that are often used in protocols for outsourcing computations (with integrity or confidentiality guarantees, or both) [49, 54, 2, 37, 31, 47, 41]. However, such constructions have so far not been explored in practice. Another powerful tool is *secure multi-party computation* [40, 13], but most work in this area does not consider the goal of succinctness.

B Case study: `memcpy`

The function `memcpy` is a standard C function that works as follows: given as input two array pointers and a length, `memcpy` copies the contents of one array to the other. Of course, with no data dependencies, copying data in a circuit is trivial: you just connect the appropriate wires. However, when the array addresses and their lengths are unknown, and `memcpy` is invoked as a subroutine in a larger program, the trivial solution *does not work*, and an efficient implementation is needed.

A naive implementation of `memcpy` iterates, via a loop, over each array position i and copies the i-th value from one array to the other. In vnTinyRAM each such loop iteration costs 6 instructions; 2 of these are to increase the iteration counter and jump back to the start of the loop. Thus, for m-long arrays, copying takes $6m$ instructions (discounting loop initialization). But, in vnTinyRAM, one can do better: loop unrolling can be used to avoid paying for the 2 "control" instructions. Asymptotically, the optimal number of unrollings *depends* on the array length: it is $\Theta(\sqrt{m})$. Thus, optimal unrolling requires dynamic code generation on a von Neumann architecture. We wrote a 54-instruction vnTinyRAM program for `memcpy` that uses dynamic loop unrolling to achieve an efficiency of $\approx 4m + 11.5\sqrt{m}$ cycles for m-long arrays. For $m \geq 600$, we get $1.25\times$ speed-up over the naive implementation, and $1.4\times$ speed-up for $m \geq 3000$.

Acknowledgments

We thank Daniel Genkin, Raluca Ada Popa, Ron Rivest, and Nickolai Zeldovich for helpful comments and discussions, and Lior Greenblatt, Shaul Kfir, Michael Riabzev, and Gil Timnat for programming assistance.

This work was supported by: the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; the Check Point Institute for Information Security; the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258; the Israeli Centers of Research Excellence I-CORE program (center 4/11); the Israeli Ministry of Science, Technology and Space; the Simons Foundation, with a Simons Award for Graduate Students in Theoretical Computer Science; and the Skolkovo Foundation.

References

[1] AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An $o(n \log n)$ sorting network. In *STOC '83* (1983).

[2] APPLEBAUM, B., ISHAI, Y., AND KUSHILEVITZ, E. From secrecy to soundness: Efficient verification via secure computation. In *ICALP '10* (2010).

[3] ARÈNE, C., LANGE, T., NAEHRIG, M., AND RITZENTHALER, C. Faster computation of the Tate pairing. *Journal of Number Theory* (2011).

[4] ARORA, S., LUND, C., MOTWANI, R., SUDAN, M., AND SZEGEDY, M. Proof verification and the hardness of approximation problems. *JACM* (1998).

[5] ARORA, S., AND SAFRA, S. Probabilistic checking of proofs: a new characterization of NP. *JACM* (1998).

[6] BABAI, L., FORTNOW, L., LEVIN, L. A., AND SZEGEDY, M. Checking computations in polylogarithmic time. In *STOC '91* (1991).

[7] BARRETO, P. S. L. M., KIM, H. Y., LYNN, B., AND SCOTT, M. Efficient algorithms for pairing-based cryptosystems. In *CRYPTO '02* (2002).

[8] BARRETO, P. S. L. M., LYNN, B., AND SCOTT, M. Efficient implementation of pairing-based cryptosystems. *Journal of Cryptology* (2004).

[9] BARRETO, P. S. L. M., AND NAEHRIG, M. Pairing-friendly elliptic curves of prime order. In *SAC'05* (2006).

[10] BEAUQUIER, B., AND ÉRIC, D. On arbitrary size Waksman networks and their vulnerability. *Parallel Processing Letters* (2002).

[11] BELLARE, M., AND GOLDREICH, O. On defining proofs of knowledge. In *CRYPTO '92* (1993).

[12] BEN-DAVID, A., NISAN, N., AND PINKAS, B. FairplayMP: a system for secure multi-party computation. In *CCS '08* (2008).

[13] BEN-OR, M., GOLDWASSER, S., AND WIGDERSON, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC '88* (1988).

[14] BEN-SASSON, E., CHIESA, A., GENKIN, D., AND TROMER, E. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *ITCS '13* (2013).

[15] BEN-SASSON, E., CHIESA, A., GENKIN, D., AND TROMER, E. On the concrete efficiency of probabilistically-checkable proofs. In *STOC '13* (2013).

[16] BEN-SASSON, E., CHIESA, A., GENKIN, D., TROMER, E., AND VIRZA, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO '13* (2013).

[17] BEN-SASSON, E., CHIESA, A., GENKIN, D., TROMER, E., AND VIRZA, M. TinyRAM architecture specification v2.00, 2013.

[18] BEN-SASSON, E., GOLDREICH, O., HARSHA, P., SUDAN, M., AND VADHAN, S. Short PCPs verifiable in polylogarithmic time. In *CCC '05* (2005).

[19] BERNSTEIN, D. J., DUIF, N., LANGE, T., SCHWABE, P., AND YANG, B.-Y. High-speed high-security signatures. In *CHES '11* (2011).

[20] BEUCHAT, J.-L., GONZÁLEZ-DÍAZ, J. E., MITSUNARI, S., OKAMOTO, E., RODRÍGUEZ-HENRÍQUEZ, F., AND TERUYA, T. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In *Pairing '10* (2010).

[21] BITANSKY, N., CANETTI, R., CHIESA, A., AND TROMER, E. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC '13* (2013).

[22] BITANSKY, N., CHIESA, A., ISHAI, Y., OSTROVSKY, R., AND PANETH, O. Succinct non-interactive arguments via linear interactive proofs. In *TCC '13* (2013).

[23] BLUM, M., DE SANTIS, A., MICALI, S., AND PERSIANO, G. Non-interactive zero-knowledge. *SIAM J. Comp.* (1991).

[24] BLUM, M., EVANS, W., GEMMELL, P., KANNAN, S., AND NAOR, M. Checking the correctness of memories. In *FOCS '91* (1991).

[25] BLUM, M., FELDMAN, P., AND MICALI, S. Non-interactive zero-knowledge and its applications. In *STOC '88* (1988).

[26] BOS, J., AND COSTER, M. Addition chain heuristics. In *CRYPTO '89* (1989).

[27] BRAUN, B., FELDMAN, A. J., REN, Z., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Verifying computations with state. In *SOSP '13* (2013).

[28] BRICKELL, E. F., GORDON, D. M., MCCURLEY, K. S., AND WILSON, D. B. Fast exponentiation with precomputation. In *EUROCRYPT '92* (1993).

[29] CANETTI, R., RIVA, B., AND ROTHBLUM, G. N. Practical delegation of computation using multiple servers. In *CCS '11* (2011).

[30] CANETTI, R., RIVA, B., AND ROTHBLUM, G. N. Two protocols for delegation of computation. In *ICITS 12* (2012).

[31] CHUNG, K.-M., KALAI, Y., AND VADHAN, S. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO '10* (2010).

[32] CORMODE, G., MITZENMACHER, M., AND THALER, J. Practical verified computation with streaming interactive proofs. In *ITCS '12* (2012).

[33] EDWARDS, H. M. A normal form for elliptic curves. *Bulletin of the American Mathematical Society* (2007).

[34] FEIGE, U., AND KILIAN, J. Making games short. In *STOC '97* (1997).

[35] FUENTES-CASTAÑEDA, L., KNAPP, E., AND RODRÍGUEZ-HENRÍQUEZ, F. Faster hashing to $\mathbb{G}_2$. In *SAC '11* (2012).

[36] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09* (2009).

[37] GENNARO, R., GENTRY, C., AND PARNO, B. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *CRYPTO '10* (2010).

[38] GENNARO, R., GENTRY, C., PARNO, B., AND RAYKOVA, M. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT '13* (2013).

[39] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC '09* (2009).

[40] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC '87* (1987).

[41] GOLDWASSER, S., KALAI, Y., POPA, R. A., VAIKUNTANATHAN, V., AND ZELDOVICH, N. Reusable garbled circuits and succinct functional encryption. In *STOC '13* (2013).

[42] GOLDWASSER, S., KALAI, Y. T., AND ROTHBLUM, G. N. Delegating computation: Interactive proofs for Muggles. In *STOC '08* (2008).

[43] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems. *SIAM J. Comp.* (1989).

[44] GRANGER, R., AND SCOTT, M. Faster squaring in the cyclotomic subgroup of sixth degree extensions. In *PKC'10* (2010).

[45] GROTH, J. Short non-interactive zero-knowledge proofs. In *ASIACRYPT '10* (2010).

[46] ISHAI, Y., KUSHILEVITZ, E., AND OSTROVSKY, R. Efficient arguments without short PCPs. In *CCC '07* (2007).

[47] KALAI, Y., RAZ, R., AND ROTHBLUM, R. Delegation for bounded space. In *STOC '13* (2013).

[48] KALAI, Y. T., AND RAZ, R. Probabilistically checkable arguments. In *CRYPTO '09* (2009).

[49] KILIAN, J. A note on efficient zero-knowledge proofs and arguments. In *STOC '92* (1992).

[50] KIM, T., KIM, S., AND CHEON, J. H. On the final exponentiation in Tate pairing computations. *IEEE Trans. on Inf. Theory* (2013).

[51] LIPMAA, H. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *TCC '12* (2012).

[52] LIPMAA, H. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *ASIACRYPT '13* (2013).

[53] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay — a secure two-party computation system. In *SSYM '04* (2004).

[54] MICALI, S. Computationally sound proofs. *SIAM J. Comp.* (2000).

[55] NAOR, M., AND YUNG, M. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *STOC '90* (1990).

[56] PARNO, B., GENTRY, C., HOWELL, J., AND RAYKOVA, M. Pinocchio: Nearly practical verifiable computation. In *Oakland '13* (2013).

[57] PIPPENGER, N. On the evaluation of powers and monomials. *SIAM J. Comp.* (1980).

[58] RIGO, A., AND PEDRONI, S. PyPy's approach to virtual machine construction. In *OOPSLA '06* (2006).

[59] SCOTT, M. Computing the Tate pairing. In *CT-RSA '05* (2005).

[60] SCOTT, M. Implementing cryptographic pairings. In *Pairing '07* (2007).

[61] SCOTT, M., AND BARRETO, P. S. L. M. Compressed pairings. In *CRYPTO '04* (2004).

[62] SCOTT, M., BENGER, N., CHARLEMAGNE, M., DOMINGUEZ PEREZ, L. J., AND KACHISA, E. J. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Pairing '09* (2009).

[63] SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Toward practical and unconditional verification of remote computations. In *HotOS '11* (2011).

[64] SETTY, S., BRAUN, B., VU, V., BLUMBERG, A. J., PARNO, B., AND WALFISH, M. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys '13* (2013).

[65] SETTY, S., MCPHERSON, M., BLUMBERG, A. J., AND WALFISH, M. Making argument systems for outsourced computation practical (sometimes). In *NDSS '12* (2012).

[66] SETTY, S., VU, V., PANPALIA, N., BRAUN, B., BLUMBERG, A. J., AND WALFISH, M. Taking proof-based verified computation a few steps closer to practicality. In *Security '12* (2012).

[67] THALER, J. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO '13* (2013).

[68] THALER, J., ROBERTS, M., MITZENMACHER, M., AND PFISTER, H. Verifiable computation with massively parallel interactive proofs. *CoRR* (2012).

[69] VALIANT, P. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC '08* (2008).

[70] VERCAUTEREN, F. Optimal pairings. *IEEE Trans. on Inf. Theory* (2010).

[71] VU, V., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. A hybrid architecture for interactive verifiable computation. In *Oakland '13* (2013).

[72] ZAHUR, S., AND EVANS, D. Circuit structures for improving efficiency of security and privacy tools. In *SP '13* (2013).

Faster Private Set Intersection based on OT Extension

Benny Pinkas
Bar-Ilan University, Israel

Thomas Schneider
TU Darmstadt, Germany

Michael Zohner
TU Darmstadt, Germany

Abstract

Private set intersection (PSI) allows two parties to compute the intersection of their sets without revealing any information about items that are not in the intersection. It is one of the best studied applications of secure computation and many PSI protocols have been proposed. However, the variety of existing PSI protocols makes it difficult to identify the solution that performs best in a respective scenario, especially since they were not all implemented and compared in the same setting.

In this work, we give an overview on existing PSI protocols that are secure against semi-honest adversaries. We take advantage of the most recent efficiency improvements in OT extension to propose significant optimizations to previous PSI protocols and to suggest a new PSI protocol whose runtime is superior to that of existing protocols. We compare the performance of the protocols both theoretically and experimentally, by implementing all protocols on the same platform, and give recommendations on which protocol to use in a particular setting.

1 Introduction

Private set intersection (PSI) allows two parties P_1 and P_2 holding sets X and Y, respectively, to identify the intersection $X \cap Y$ without revealing any information about elements that are not in the intersection. The basic PSI functionality can be used in applications where two parties want to perform JOIN operations over database tables that they must keep private, e.g., private lists of preferences, properties, or personal records of clients or patients. PSI is used for privacy-preserving computation of functionalities such as relationship path discovery in social networks [37], botnet detection [40], testing of fully-sequenced human genomes [3], proximity testing [43], or cheater detection in online games [10]. Another use case is measurement of the performance of web ad campaigns, by comparing purchases by users who were shown a specific ad to purchases of users who were not shown the ad. This is essentially a variant of PSI where the input of the web advertising party is the identities of the users who were shown the ad, and the input of the merchant, or of an agency that operates on its behalf, is the identities of the buyers. It was published that Facebook and Datalogix, a consumer data collection company, perform this type of measurements.[1] (The article indicates that they seem to be using the insecure hash-based solution described in §1.1, but instead they can use a properly secure PSI protocol while still being reasonably efficient.)

PSI has been a very active research field, and there have been many suggestions for PSI protocols. The large number of proposed protocols makes it non-trivial to perform comprehensive cross-evaluations. This is further complicated by the fact that many protocol designs have not been implemented and evaluated, were analyzed under different assumptions and observations, and were often optimized w.r.t. overall runtime while neglecting other relevant factors such as communication.

In this paper, we give an overview on existing efficient PSI protocols, optimize the recently proposed PSI protocols of [27] and [17], based on garbled circuits and Bloom filters, respectively, and describe a new PSI protocol based on recent results in the area of efficient OT extensions [1, 35]. We compare both the theoretical and empirical performance of all protocols on the same platform and conclude with remarks on the protocols and their suitability for different scenarios.

1.1 Classification of PSI Protocols

A naive solution When confronted with the PSI problem, most novices come up with a solution where both parties apply a cryptographic hash function to their inputs and then compare the resulting hashes. Although this protocol is very efficient, it is insecure if the input domain is not large or does not have high entropy, since one party could easily run a brute force attack that applies the hash function to all items that are likely to be in the input set and compare the results to the received hashes. (When inputs to PSI have a high entropy, a protocol that compares hashes of the inputs can be used [41].)

[1] https://www.eff.org/deeplinks/2012/09/deep-dive-facebook-and-datalogix-whats-actually-getting-shared-and-how-you-can-opt

PSI is one of the best studied problems in secure computation. Since its introduction, several techniques have been used to realize PSI protocols. While the first PSI protocols were special-purpose solutions based on public-key primitives, other solutions were based on circuit-based generic techniques of secure computation, that are mostly based on symmetric cryptography. A recent development are PSI protocols that are based on oblivious transfer (OT) alone, and combine the efficiency of symmetric cryptographic primitives with special purpose optimizations. Finally, we describe PSI protocols that utilize a third party to achieve even better efficiency.

Public-Key-Based PSI A PSI protocol based on the Diffie-Hellmann (DH) key agreement scheme was presented in [29] (related ideas were presented earlier in [36]). This protocol is based on the commutative properties of the DH function and was used for private preference matching, which allows two parties to verify if their preferences match to some degree.

Freedman et al. [21] introduced PSI protocols secure against semi-honest and malicious adversaries in the standard model (rather than in the random oracle model assumed in the DH-based protocol). This protocol was based on polynomial interpolation, and was extended in [19], which presents protocols with simulation-based security against malicious adversaries, and evaluates the practical efficiency of the proposed hashing schemes. We discuss the proposed hashing schemes in §6. A similar approach that uses oblivious pseudo-random functions to perform PSI was presented in [20]. A protocol that uses polynomial interpolation and differentiation for finding intersections between multi-sets was presented in [34].

Another PSI protocol that uses public-key cryptography (more specifically, blind-RSA operations) and scales linearly in the number of elements was presented in [14] and efficiently implemented and benchmarked in [15].

A PSI protocol based on additively homomorphic encryption was described in [11], but is excluded from this evaluation since it scales quadratically in the number of elements and is hence slower than related solutions.

Circuit-Based PSI Generic secure computation protocols have been subject to huge efficiency improvements in the last decade. They allow the secure evaluation of arbitrary functions, expressed as Arithmetic or Boolean circuits. Several Boolean circuits for PSI were proposed in [27] and evaluated using the Yao's garbled circuits framework of [28]. The authors showed that their Java implementation scales very well with increasing security parameter and outperforms the blind-RSA protocol of [14] for larger security parameter.[2] We reflect on and present new optimizations for circuit-based PSI in §3.

OT-Based PSI A recent PSI protocol of [17] uses Bloom filters [9] and OT extension [30] to obtain very efficient PSI protocols with security against semi-honest and malicious adversaries. We describe this protocol and our optimization using random OT extension [1] in §4.

Third Party-Based PSI Several PSI protocols have been proposed that utilize additional parties, e.g., [4]. In [25], a trusted hardware token is used to evaluate an oblivious pseudo-random function. This approach was extended to multiple untrusted hardware tokens in [18]. Several efficient server-aided protocol for PSI were presented and benchmarked in [32]. For their PSI protocol with a semi-honest server, the authors report a runtime of 1.7 s for server-aided PSI on one million elements using 20 threads between cloud instances in the US east - and west coast and 10 MB of communicated data. In comparison, our fastest PSI protocol without a server requires 4.9 s for 2^{18} elements using four threads and sends 78 MB (cf. Tab. 1 and Tab. 8). Note that this comparison is sketchy and is only meant to demonstrate that using a third party can increase performance. In our work we focus on PSI protocols without a third party.

1.2 Our Contributions

We describe in detail the PSI protocols based on generic secure computation and on Bloom filters, and suggest how to improve their performance using carefully analyzed features of OT extension. We then introduce a new OT-based PSI protocol, and perform a detailed experimental comparison of all the PSI protocols that we described. In the following, we detail our contributions.

Optimizations of Existing Protocols We improve the circuit- and Bloom-filter-based PSI protocols using recent optimizations for OT extension [1]. In particular, in §3 we evaluate the circuit-based solution of [27] on a secure evaluation of the GMW protocol, and utilize features of random OT (cf. §2.2) to optimize the performance of multiplexer gates (which form about two thirds of the circuit). In §4.3 we redesign the Bloom filter-based protocol of [17] to benefit from using random OT and to support multi-core environments.

[2] Subsequent work of [15] claimed that the blind-RSA protocol of [14] runs faster than the circuit-based protocol of [27] even for larger security parameter. Their implementation is in C++ instead of Java.

A Novel OT-Based PSI Protocol We present a new PSI protocol that is directly based on OT (§5) and directly benefits from recent improvements in efficient OT extensions [1, 35]. The basic version of the protocol can efficiently compare one element with many elements, but for PSI on n elements it requires $O(n^2 \log n)$ communication. In §6 we use carefully analyzed hashing techniques in order to achieve $O(n \log n)$ communication. The resulting protocol has very low computation complexity since it mostly requires symmetric key operations and has even less communication than some public-key-based PSI protocols.

A Detailed Comparison of PSI Protocols We implement the most promising candidate PSI protocols using state-of-the-art cryptographic techniques and compare their performance on the same platform. As far as we know, this is the first time that such a wide comparison has been made, since previous comparisons were either theoretical, compared implementations on different platforms or programming languages, or used implementations without state-of-the-art optimizations. Our implementations and experiments are described in detail in §7. Certain experimental results were unexpected. We give a partial summary of our results in Tab. 1: the values in parenthesis give the overhead of the original protocols and highlight the gains achieved by our optimizations.

PSI Protocol	DH ECC [29]	Circuit [27] optimized GMW §3.2 (original GMW [1])	Bloom Filter optimized §4.3 (original [17])	OT §5+§6
Runtime (s)	416	762 (1,304)	68 (154)	14
Comm. (MB)	24	14,040 (23,400)	740 (1,393)	78

Table 1: Runtime and transferred data for private set intersection protocols on sets with 2^{18} 32-bit elements and 128-bit security with a single thread over Gigabit LAN.

We highlight here the conclusions of our results:

- The Diffie-Hellman-based protocol [29], which was the first PSI protocol, is actually the most efficient w.r.t. communication (when implemented using elliptic-curve crypto). Therefore it is suitable for settings with distant parties which have strong computation capabilities but limited connectivity.
- Generic circuit-based protocols [27] are less efficient than the newer, OT-based constructions, but they are more flexible and can easily be adapted for computing variants of the set intersection functionality (e.g., computing whether the size of the intersection exceeds some threshold). Our experiments also support the claim of [27] that circuit-based PSI protocols are faster than the blind-RSA-based PSI protocol of [14] for larger security parameters and given sufficient bandwidth.
- While for larger security parameter previously proposed circuit- and OT-based protocols can be faster than the public-key-based protocols on a Gigabit LAN, the DH-based protocol of [29] outperforms *all* of them in an Internet network setting. Our new OT-based protocol (§5+§6) is the only protocol that maintains its performance advantage in this setting and even outperforms public-key-based PSI protocols for a mobile network setting.

2 Preliminaries

We give our notation and security definitions in §2.1 and review recent relevant work on oblivious transfer in §2.2.

2.1 Notation and Security Definitions

We denote the parties as P_1 and P_2, and their respective input sets as X and Y with $|X| = n_1$ and $|Y| = n_2$. When the two input sets are of equal size, we use $n = n_1 = n_2$. We refer to elements from X as x and elements from Y as y and each element has bit-length σ (we detail the relation between n and σ in the full version [47]).

We write $b[i]$ for the i-th element of a list b, denote the bitwise-AND between two bit strings a and b of equal length as $a \wedge b$ and the bitwise-XOR as $a \oplus b$.

We refer to a correlation resistant one-way function as CRF, and to a pseudo-random generator as PRG.

We write $\binom{N}{1}$-OT_ℓ^m for m parallel 1-out-of-N oblivious transfers on ℓ-bit strings, and write OT_ℓ^m for $\binom{2}{1}$-OT_ℓ^m.

Security parameters We denote the symmetric security parameter as κ, the asymmetric security parameter as ρ, the statistical security parameter as λ, and use the recommended key sizes of the NIST guideline [45], summarized in Tab. 2. We denote the bit size of elliptic curve points with φ, i.e., $\varphi = 284$ for Koblitz curve K-283 when using point compression.

Security	SYM (κ)	FFC and IFC (ρ)	ECC (φ)	Hash
80-bit	80	1,024	K-163	SHA-1
128-bit	128	3,072	K-283	SHA-256

Table 2: NIST recommended key sizes for symmetric cryptography (SYM), finite field cryptography (FFC), integer factorization cryptography (IFC), elliptic curve cryptography (ECC) and hash functions.

Adversary definition The secure computation literature considers two types of adversaries with different strengths: A *semi-honest adversary* tries to learn as much information as possible from a given protocol execution but is not able to deviate from the protocol steps. The semi-honest adversary model is appropriate for scenarios where software attestation is enforced or where an untrusted third party is able obtain the transcript of the protocol after its execution, either by stealing it or by legally enforcing its disclosure. The stronger, *malicious adversary* extends the semi-honest adversary by being able to deviate arbitrarily from the protocol steps.

Most protocols for private set intersection, as well as this work, focus on solutions that are secure against semi-honest adversaries. PSI protocols for the malicious setting exist, but they are considerably less efficient than protocols for the semi-honest setting (see, e.g., [13, 16, 19, 21, 26, 31]).

The random oracle model As most previous works on efficient PSI, we use the random oracle model to achieve more efficient implementations [8]. We provide details and argue about the use of random oracles in the full version [47].

2.2 Oblivious Transfer

Oblivious transfer (OT) is a major building block for secure computation. When executing m invocations of 1-out-of-2 OT on ℓ-bit strings (denoted $\binom{2}{1}$-OT_ℓ^m), the sender S holds m message pairs (x_0^i, x_1^i) with $x_0^i, x_1^i \in \{0,1\}^\ell$, while the receiver R holds an m-bit choice vector b. At the end of the protocol, R receives $x_{b[i]}^i$ but learns nothing about $x_{1-b[i]}^i$, and S learns nothing about b. Many OT protocols have been proposed, most notably (for the semi-honest model) the Naor-Pinkas OT [42], which uses public-key operations and has amortized complexity of $3m$ public-key operations when performing m OTs.

OT extension [6, 30] reduces the number of expensive public-key operations for OT_ℓ^m to that of only $\mathrm{OT}_\kappa^\kappa$, and computes the rest of the protocol using more efficient symmetric cryptographic operations which are orders of magnitude faster. The security parameter κ is essentially independent of the number of OTs m, and can be as small as 80 or 128. Thereby, the computational complexity for performing OT is reduced to such an extent, that the network bandwidth becomes the main bottleneck [1].

Recently, the efficiency of OT extension protocols has gained a lot of attention. In [35], an efficient 1-out-of-N OT extension protocol was shown, that has sub-linear communication in κ for short messages. Another protocol improvement is outlined in [1, 35], which decreases the communication from R to S by half. Additionally, several works [1, 44] improve the efficiency of OT by using an OT variant, called **random OT**. In random OT, (x_0^i, x_1^i) are chosen uniformly and randomly within the OT and are output to S, thereby removing the final message from S to R. Random OT is useful for many applications, and we show how it can reduce the overhead of PSI. We elaborate on these OT extension protocols in the full version [47].

3 Circuit-Based PSI

Unlike special purpose private set intersection protocols, the protocols that we describe in this section are based on a *generic* secure computation protocol that can be used for computing arbitrary functionalities. State-of-the-art for computing the PSI functionality is the sort-compare-shuffle (SCS) circuit of [27], which has size $\mathcal{O}(n \log n)$ (cf. full version [47]for details.) We discuss these protocols by reflecting on the generic secure computation protocol of Goldreich-Micali-Wigderson (GMW) [23] (§3.1) and outlining major optimizations for evaluating the SCS circuit for PSI using GMW (§3.2).

The usage of generic protocols holds the advantage that the functionality of the protocol can easily be extended, without having to change the protocol or the security of the resulting protocol. For example, it is straightforward to change the protocol to compute the size of the intersection, or a function that outputs 1 iff the intersection is greater than some threshold, or compute a summation of values (e.g., revenues) associated with the items that are in the intersection. Computing these variants using other PSI protocols is non-trivial.

3.1 The GMW Protocol

We focus on the GMW protocol [23] for generic secure computation, which was implemented in the semi-honest model for multiple parties in [12], optimized for two parties in [49], and extended to the malicious model in [44].

The GMW protocol represents the function to be computed as a Boolean circuit and uses an XOR-based secret-sharing and OT to evaluate the circuit. A circuit with input bit u from P_1 and v from P_2 is evaluated as follows. First, P_1 and P_2 secret-share their input bit $u = u_1 \oplus u_2$ and $v = v_1 \oplus v_2$ and P_i obtains the shares labeled with i. The parties then evaluate the Boolean circuit gate-by-gate, as detailed next. To evaluate an XOR gate with input wires u and v and output wire w, P_1 locally computes $w_1 = u_1 \oplus v_1$ while P_2 locally computes $w_2 = u_2 \oplus v_2$.

Evaluating AND gates using multiplication triples An AND gate with input wire u and v and output wire w

requires an interaction between both parties using a *multiplication triple* [5]. A multiplication triple is a set of shares $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2 \in \{0,1\}$ with $(\alpha_1 \oplus \alpha_2) \wedge (\beta_1 \oplus \beta_2) = \gamma_1 \oplus \gamma_2$. Given a multiplication triple, to evaluate an AND gate implementing u AND v, the parties compute $d_i = \alpha_i \oplus u_i$ and $e_i = \beta_i \oplus v_i$, exchange d_i, e_i, reconstruct $d = d_1 \oplus d_2$ and $e = e_1 \oplus e_2$, and compute the shares of the gate output wire as $w_1 = (d \wedge e) \oplus (d \wedge \beta_1) \oplus (e \wedge \alpha_1) \oplus \gamma_1$ and $w_2 = (d \wedge \beta_2) \oplus (e \wedge \alpha_2) \oplus \gamma_2$. These are all extremely efficient operations and therefore the efficiency of the evaluation depends on the efficiency of generating multiplication triples.

As described in [1], multiplication triples can be generated using two random OTs on one-bit strings as follows: both parties choose $\alpha_i \in_R \{0,1\}$ and run two random OTs, where in the first OT P_1 acts as sender and P_2 as receiver with choice bit α_2, and in the second OT P_2 acts as sender and P_1 as receiver with choice bit α_1. From each OT, the sender obtains (x_0^i, x_1^i) and sets $\beta_i = x_0^i \oplus x_1^i$ and the receiver obtains $x_{\alpha_i}^i$. To compute valid γ_0, γ_1 values for the triple, note that $(\alpha_1 \oplus \alpha_2) \wedge (\beta_1 \oplus \beta_2) = (\alpha_1 \wedge \beta_1) \oplus (\alpha_1 \wedge \beta_2) \oplus (\alpha_2 \wedge \beta_1) \oplus (\alpha_2 \wedge \beta_2) = \gamma_0 \oplus \gamma_1$. P_i locally computes $\alpha_i \wedge \beta_i$. Values $\alpha_1 \wedge \beta_2$ and $\alpha_2 \wedge \beta_1$ are computed using the output of the random OT as $\alpha_1 \wedge \beta_2 = x_{\alpha_1}^2 \oplus x_0^2$ and $\alpha_2 \wedge \beta_1 = x_{\alpha_2}^1 \oplus x_0^1$. Finally, P_1 sets $\gamma_1 = (\alpha_1 \wedge \beta_1) \oplus x_0^1 \oplus x_{a_1}^2$ and P_2 sets $\gamma_2 = (\alpha_2 \wedge \beta_2) \oplus x_0^2 \oplus x_{\alpha_2}^1$. These computations can be done in a preprocessing step before the input is known, are independent of circuit's structure, and highly parallelizable.

3.2 Optimized Circuit-Based PSI

We describe in this section an optimization which greatly reduces the overhead of circuit based PSI for GMW (as is detailed in Tab. 5 in §7, the reduction in the runtime for inputs of size 2^{18} is about 40%). The optimization is based on a protocol proposed in [39].

Approximately $2/3$ of the AND gates in the SCS circuit are due to multiplexers (cf. full version [47] for details). In each multiplexer operation with σ-bit inputs x and y and a choice bit s, we compute $z[j] = s \wedge (x[j] \oplus y[j]) \oplus x[j]$ for each $1 \leq j \leq \sigma$ using σ AND gates in total. The evaluation of this multiplexer circuit in the GMW protocol requires random $\mathrm{OT}_1^{2\sigma}$, namely 2σ random OTs of single-bit inputs. We observe that the same wire s is input to multiple AND gates which allows for the following optimization.

Consider an input wire u that is the input to multiple AND gates of the form $w[1] = (u \text{ AND } v[1]), \ldots, w[\sigma] = (u \text{ AND } v[\sigma])$. Similar to the evaluation of a single AND gate described in §3.1, these gates can be evaluated using a multiplication triple generalized to vectors, which we call a *vector multiplication triple*.

A vector multiplication triple has the following form: $\alpha_1, \alpha_2 \in \{0,1\}$; $\beta_1, \beta_2, \gamma_1, \gamma_2 \in \{0,1\}^\sigma$, where P_i holds the shares labeled with i that satisfy the condition $(\alpha_1 \oplus \alpha_2) \wedge (\beta_1[j] \oplus \beta_2[j]) = \gamma_1[j] \oplus \gamma_2[j]$. To evaluate the AND gates, both parties compute $d_i = \alpha_i \oplus u_i$ and $e_i[j] = \beta_i[j] \oplus v_i[j]$, exchange $d_i, e_i[j]$, set $d = d_1 \oplus d_2$, $e[j] = e_1[j] \oplus e_2[j]$, and $w_i[j] = (d \wedge e[j]) \oplus (d \wedge \beta_i[j]) \oplus (e[j] \wedge \alpha_i) \oplus \gamma_i[j]$. The vector multiplication triple can be pre-computed analogously to the regular multiplication triples described in §3.1, but using random OT_σ^2, namely only two random OTs applied to σ-bit strings: The parties each choose $\alpha_1, \alpha_2 \in_R \{0,1\}$ and perform a random OT_σ^1 with P_1 acting as sender and P_2 acting as receiver with choice bit α_2, and a second random OT_σ^1 with P_2 acting as sender and P_1 acting as receiver with choice bit α_1. From these random OTs, P_i obtains $\beta_i \in \{0,1\}^\sigma = x_0^i \oplus x_1^i$ and, analogously to the regular multiplication triple generation, a valid $\gamma_i \in \{0,1\}^\sigma$.

Efficiency Overall, evaluating σ AND gates with a vector multiplication triple requires to send $2\sigma + 2$ bits (instead of 4σ bits with σ regular multiplication triples). Generating a vector multiplication triple requires 2 random OTs on σ-bit strings (instead of 2σ random OTs with σ regular multiplication triples); as the communication of random OT is independent of the input length, this improves communication by factor σ.

In the SCS circuit we have $2n\log_2 n + n + 1$ multiplexers, each of which can be evaluated using a single vector multiplication triple. This reduces the number of random OTs from $2\sigma(2n\log_2 n + n + 1)$ to $2(2n\log_2 n + n + 1)$.

Further applications of vector multiplication triples As a side note, we comment that our vector multiplication triples can be used in every circuit where wires are used as input in two or more AND gates. As such, another beneficial application is multiplication. When computing a multiplication between two σ-bit numbers x and y using the school method multiplication circuit [49], each bit x_i is multiplied with every bit of y: $\forall_{1 \leq i \leq \sigma} \forall_{1 \leq j \leq \sigma} (x_i \wedge y_j)$. Here, using vector multiplication triples allows to reduce the total number of random OTs by a factor two, from $4\sigma^2 - 2\sigma$ OTs to $2\sigma^2$.

4 Bloom Filter-Based PSI

The recent PSI protocol of [17] uses Bloom Filters (BF) and OT to compute set intersection. We summarize Bloom filters in §4.1 and the PSI protocol of [17] in §4.2. We then present a redesigned optimized version of the protocol in §4.3. This optimization reduces the runtime for inputs of size 2^{18} by $55\% - 60\%$ (cf. §7, Tab. 5).

4.1 The Bloom Filter

A BF that represents a set of n elements consists of an m-bit string F and k independent uniform hash functions $h_1,...,h_k$ with $h_i : \{0,1\}^* \mapsto [1,m]$, for $1 \leq i \leq k$. Initially, all bits in F are set to zero. An element x is inserted into the BF by setting $F[h_i(x)] = 1$ for all i. To query if the BF contains an item y, one checks all bits $F[h_i(y)]$. If there is at least one j such that $BF[h_j(y)] = 0$, then y is not in the BF. If, on the other hand, all bits $BF[h_i(y)]$ are set to one, then y is in the BF except for a false positive probability ε. An upper bound on ε can be computed as $\varepsilon = p^k(1+O(\frac{k}{p}\sqrt{\frac{\ln m - k\ln p}{m}}))$, where $p = 1-(1-\frac{1}{m})^{kn}$. The authors of [17] propose to choose the number of hash functions as $k = 1/\varepsilon$ and the size of the BF as $m = kn/\ln 2 \approx 1.44kn$. In their experiments, they set $\varepsilon = 2^{-\kappa}$, resulting in $k = \kappa$ and a filter of size $m \approx 1.44\kappa n$.

4.2 Garbled Bloom Filter-Based PSI

For BF-based PSI, one cannot simply compute the bitwise AND of the BFs that represent each set, as this leaks information (see [17] for details). Instead, the authors of [17] introduced a variant of the BF, called Garbled Bloom Filter (GBF). Like a BF, a GBF G uses κ hash functions $h_1,...,h_\kappa$, but instead of single bits, it holds shares of length ℓ at each position $G[i]$, for $1 \leq i \leq m$. These shares are chosen uniformly at random, subject to the constraint that for every element x contained in the filter G it holds that $\bigoplus_{j=1}^{\kappa} G[h_j(x)] = x$.

To represent a set X using a GBF G, all positions of G are initially marked as unoccupied. Each element $x \in X$ is then inserted as follows. First, the insertion algorithm tries to find a hash function $t \in [1...\kappa]$ such that $G[h_t(x)]$ is unoccupied (the probability of not finding such a function is equal to the probability of a false positive in the BF, which is negligible due to the choice of parameters). All other unoccupied positions $G[h_j(x)]$ are set to random ℓ-bit shares. Finally, $G[h_t(x)]$ is set to $G[h_t(x)] = x \oplus \left(\bigoplus_{j=1, j\neq t}^{\kappa} G[h_j(x)]\right)$ to obtain a valid sharing of x. We emphasize that because existing shares need to be re-used, the generation of the GBF cannot be fully parallelized. (We describe below in §4.3 how the protocol can be modified to enable a parallel execution.)

In the semi-honest secure PSI protocol of [17], P_1 generates a m-bit GBF G_X from its set X and P_2 generates a m-bit BF F_Y from its set Y. P_1 and P_2 then perform OT_ℓ^m, where for the i-th OT P_1 acts as a sender with input $(0, G_X[i])$ and P_2 acts as a receiver with choice bit $F_Y[i]$. Thereby, P_2 obtains an intersection GBF $G_{(X\wedge Y)}$, for which $G_{(X\wedge Y)}[i] = 0$ if $F_Y[i] = 0$ and $G_{(X\wedge Y)}[i] = G_X[i]$ if $F_Y[i] = 1$. P_2 can check whether an element y is in the intersection by checking whether $\bigoplus_{i=1}^{k} G_{(X\wedge Y)}[h_i(y)] \stackrel{?}{=} y$. (Note that P_2 cannot perform this check for any value which is not in its input set, since the probability that it learns all GBF locations associated with that value is equal to the probability of a false positive, which is negligible due to the choice of parameters.) The bit-length of the shares in the GBF can be set to $\ell = \lambda$.

4.3 Random GBF-Based PSI

We introduce an optimization of the GBF-based PSI protocol of [17], which we call the *random Garbled Bloom Filter* protocol. The core idea is to have parties collaboratively generate a *random* GBF. This is in contrast to the original protocol where the GBF had to be of a specific structure (i.e., have the XOR of the entries of $x \in X$ be x). The modified protocol can be based on random OT extension (in fact, on a version of the protocol which is even more efficient than the original random OT extension). For each position in the filter, each party learns a random value if the corresponding bit in its BF is 1. P_1 then sends to P_2 the XOR of the GBF values corresponding to each of its inputs, and P_2 compares these values to the XOR of the GBF values of its own inputs.

We denote the primitive that enables this solution an oblivious pseudo-random generator (OPRG), which takes as inputs bits b_1, b_2 from each party, respectively, generates a random string s, and outputs to P_t s if $b_t = 1$ and nothing otherwise, for $t \in \{0,1\}$. Additionally, we require that the parties remain oblivious to whether the other party obtained s. A protocol for computing this functionality is obtained by modifying the existing random OT extension protocol of [1] as follows.

Recall that in random OT extension, S has no input in the i-th OT and outputs two values (x_0^i, x_1^i), while R inputs a choice bit vector b and outputs $x_{b[i]}^i$. Computation of each of these values involves one evaluation of a hash function H (cf. §2.2; the detailed random OT extension protocol is summarized in the full version [47]). The new functionality is obtained by having S ignore the x_0^i output that it receives, and ignore also the x_1^i output if $b_1 = 0$. Similarly, R ignores its output if $b_2 = 0$. The random OT extension protocol thus becomes more efficient, since the parties can ignore parts of the computation.

Our resulting Bloom filter-based protocol works as follows. First, P_1 and P_2 each generate a BF, F_X and F_Y respectively. They evaluate the OPRG with P_1 being the sender and P_2 being the receiver, using the bits of F_X and F_Y as inputs, to obtain random GBFs G_X and G_Y with entries in $\{0,1\}^\ell$. For each element x_j in its set X, P_1 then computes $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} G_X[h_i(x_j)]$, with $1 \leq j \leq n_1$. Finally, P_1 sends all m_{P_1} values in random order to P_2, which identifies whether an element y in its set is in the intersection by checking whether a j exists such that $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} G_Y[h_i(y)]$.

Correctness For each item in the intersection, P_2 gets from P_1 the same XOR value that it computed from its own GBF, and therefore identifies that the item is in the intersection. For any item which is not in the intersection, it holds with overwhelming probability that the XOR value computed by P_2 is independent of the n_1 values received from P_1. The probability of a false positive identification for that value is therefore $n_1 \cdot 2^{-\ell}$. The probability of a false positive identification for any of the values is $n_1 n_2 \cdot 2^{-\ell}$. To achieve correctness with probability 1-$2^{-\lambda}$, we therefore set $\ell = \lambda + \log_2 n_1 + \log_2 n_2$.

Security The security of each party can be easily proved using a simulation argument. P_2's security is obvious, since the only information that P_1 learns are the random outputs of the random OT protocol, which are independent of P_2's input and can be easily simulated by P_1. P_1's security is apparent from observing that the information that P_2 receives from P_1 is composed of

- The XOR values that P_2 computed for each item in the intersection.
- The XOR values that P_1 computed for its $n_1 - |X \cap Y|$ items that are not in the intersection. These values are independent of P_2's BF unless one of these items is a false-positive identification in the filter, which happens with negligible probability ε.

Therefore, the information received from P_1 can be easily simulated by P_2 given its legitimate output, i.e., $X \cap Y$.

Efficiency As shown in Tab. 3, our resulting random GBF-based PSI protocol has less computation and communication complexity than the original GBF protocol in [17]. In terms of communication, in our new protocol, P_1 has to send the $n_1\ell$-bit vector m_{P_1} and P_2 has to send $m\kappa$ bits in the random OTs. (This is compared to $2m\lambda$ bits and $2m\kappa$ bits sent in the original protocol. Later in our experiments in §7 we show that the communication is reduced by a factor between 1.9 and 3, cf. Tab. 6.)

The computation complexity of our protocol is $\mathrm{HW}(F_X)$ hash function evaluations for P_1 and $\mathrm{HW}(F_Y)$ hash function evaluations for P_2, where $\mathrm{HW}(\cdot)$ denotes the Hamming weight. When the number of hash functions k and the size of the BF m are chosen optimally, we can approximate the average Hamming weight in a BF using the probability that a single bit is set to 1, which is $1-(1-(\frac{1}{m}))^{kn} \approx \frac{1}{2}$. Thus, $\mathrm{HW}(F) \approx \frac{m}{2}$.

A main advantage of our protocol is that it allows to parallelize *all* operations: BFs can be generated in parallel (bits in the BF are changed only from 0 to 1) and, most importantly, the random GBF can also be constructed in parallel, in contrast to the original GBF-based protocol.

Optimization	Party	# Bits Sent	# calls to H
Original GBF-based PSI [17]	P_1	$2m\lambda$	$2m$
	P_2	$2m\kappa$	m
Random GBF-based PSI (§4.3)	P_1	$n_1\ell$	$m/2$
	P_2	$m\kappa$	$m/2$

Table 3: Communication and computation complexities for Bloom-filter-based PSI of [17] and our optimization. (λ: statistical security parameter, κ: symmetric security parameter, n_i: number of elements of party P_i, $m \approx 1.44\kappa\max(n_1,n_2)$, $\ell = \lambda + \log_2 n_1 + \log_2 n_2$).

5 Private Set Intersection via OT

We propose a new private set intersection protocol that is based on the most efficient OT extension techniques, in particular the random OT functionality [1, 44] and the efficient 1-out-of-N OT of [35]. This PSI protocol scales very efficiently with an increasing set size.

We first describe the protocol for a private equality test (PEQT) between two elements x and y (§5.1) and then describe how to efficiently extend it for comparing y to a set $X = \{x_1, ..., x_n\}$ (§5.2). The resulting protocol can then be simply extended to perform PSI between sets X and Y by applying the parallel comparison protocol for each element $y \in Y$ (§5.3). The overhead of the protocol can be greatly improved using hashing (§6).

5.1 The Basic PEQT Protocol

In the most basic private equality test (PEQT) protocol, P_1 and P_2 check whether their σ-bit elements x and y are equal by engaging in random $\binom{2}{1}$ $\mathrm{OT}^{\sigma}_{\ell}$, where P_2 uses the bits of y as its choice vector. From each random OT, P_1 obtains two uniformly distributed and random ℓ-bit strings (s^i_0, s^i_1), and P_2 obtains $s^i_{y[i]}$. P_1 then computes $m_{P_1} = \bigoplus_{i=1}^{\sigma} s^i_{x[i]}$ (the XOR of the strings corresponding to the binary representation of x) and sends it to P_2. P_2 compares this value to $m_{P_2} = \bigoplus_{i=1}^{\sigma} s^i_{y[i]}$ and decides that $x = y$ iff $m_{P_1} = m_{P_2}$.

The basic private equality test can be improved by using a base-N representation of the inputs and a $\binom{N}{1}$ OT in the protocol. Specifically, let $N = 2^{\eta}$. P_1 and P_2 check whether their σ-bit elements x and y are equal by representing them using $t = \sigma/\eta$ letters from an alphabet of size N, and then engaging in random $\binom{N}{1}$-OT^t_{ℓ}.

For this, P_2 cuts its σ-bit element y into t blocks $y[i]$ of bitlength η each: $y = y[1]||\ldots||y[t]$; similarly, P_1 interprets $x = x[1]||\ldots||x[t]$. In the i-th random $\binom{N}{1}$-OT, P_2 inputs $y[i]$ as choice bits and P_1 obtains N random and uniformly distributed ℓ-bit strings $(s^i_0, ..., s^i_{N-1})$; P_2 obtains $s^i_{y[i]}$. P_1 sends $m_{P_1} = \bigoplus_{i=1}^{t} s^i_{x[i]}$ to P_2 who compares it to $m_{P_2} = \bigoplus_{i=1}^{t} s^i_{y[i]}$ and decides that $x = y$ iff $m_{P_1} = m_{P_2}$.

Correctness If $x = y$ then the choices that both parties make for their sums are equal, i.e., $m_{P_1} = \bigoplus_{i=1}^{t} s^i_{x[i]} = \bigoplus_{i=1}^{t} s^i_{y[i]} = m_{P_2}$, and P_2 successfully identifies equality.

If $x \neq y$ then the probability that $m_{P_1} = m_{P_2}$ is $2^{-\ell}$. To see that this is true, assume w.l.o.g. that the inputs differ in their last sub-string, i.e., $x[t] \neq y[t]$. Equality only holds if the last element received by P_2, namely $s^t_{y[t]}$, is equal to $\bigoplus_{i=1}^{t} s^i_{x[i]} \oplus \bigoplus_{i=1}^{t-1} s^i_{y[i]}$. The value of $s^t_{y[t]}$ is independent of the other values, and therefore this equality happens with probability $2^{-\ell}$ and thus we can set ℓ to be equal to the statistical security parameter λ.

Security P_2's security is obvious, since the only information that P_1 learns are the random values chosen in the random OT, which are independent of P_2's input.

As for P_1's security, note that P_2's view in the protocol consists of its t outputs in the random $\binom{N}{1}$-OT protocols, and of the value m_{P_1} sent by P_1. If $x = y$ then m_{P_1} is equal to the XOR of the first t values. Otherwise, all $t+1$ values are uniformly distributed. In both cases, the view of P_2 can be easily simulated given the output of the protocol (i.e., knowledge whether $x = y$). The protocol is therefore secure according to the common security definitions of secure computation [22].

Efficiency Since in the i-th random OT P_1 needs only the output $s^i_{x[i]}$, it suffices to evaluate one hash function per random OT. When using the random $\binom{2}{1}$-OT extension protocol[3] of [1] and $\ell = \lambda$, the parties perform random $\text{OT}^{\sigma}_{\lambda}$, send $\sigma\kappa + \lambda$ bits, and do σ hash function evaluations each. In comparison, when using the random $\binom{N}{1}$-OT extension protocol of [35], the parties perform only σ/η OTs and send 2κ bits per OT; in total, they have to send $2\sigma\kappa/\eta + \lambda$ bits and do σ/η hash function evaluations each. In the full version [47] we provide an analysis which shows that setting $\eta = 8$ results in optimal performance for our PSI protocols.

5.2 Private Set Inclusion Protocol

In a private set inclusion protocol, P_1 and P_2 check whether y equals any of the values in $X = \{x_1, ..., x_{n_1}\}$. The set inclusion protocol is similar to the basic PEQT protocol, but in order to perform multiple comparisons in parallel, the OTs are computed over longer strings, essentially transferring (in parallel) a random string for each element in the set X.

In more detail, both parties run a random $\binom{N}{1}$-$\text{OT}^t_{n_1\ell}$, where P_2 uses the bits of y as choice bits. Each received string is of length $n_1\ell$ bits. That is, in the i-th random OT, P_1 obtains N random strings $(s^i_0, ..., s^i_{N-1}) \in \{0,1\}^{n_1\ell}$, and P_2 obtains one random string $s^i_{y[i]}$. The strings are parsed as a list of n_1 sub-strings of length ℓ bits each. We refer to the j-th sub-string in these lists as $s^i_w[j]$, for $1 \leq j \leq n_1$ and $0 \leq w < N$. Using these sub-strings, P_1 and P_2 can then compute the XOR of the strings corresponding to their respective inputs, compare the results and decide on equality, as was described in the basic PEQT protocol in §5.1. In more detail, P_1 computes $m_{P_1}[j] = \bigoplus_{i=1}^{t} s^i_{x_j[i]}[j]$ and sends the $n_1\ell$-bit string m_{P_1} to P_2. P_2 decides whether y matches any of the elements in X by computing $m_{P_2} = \bigoplus_{i=1}^{t} s^i_{y[i]}$ and checking whether a j exists with $m_{P_1}[j] = m_{P_2}$.

Correctness and security follow from the properties of the protocol of §5.1. However, now we require that the value m_{P_2} and *all* the n_1 values $m_{P_1}[j]$ are distinct, which happens with probability $n_1 2^{-\ell}$. Thus, to achieve correctness with probability 1-$2^{-\lambda}$, we must increase the bit-length of the OTs to $\ell = \lambda + \log_2 n_1$. Also, note that P_2 learns the position j at which the match is found, which can be avoided by randomly permuting the inputs.

Efficiency The set inclusion protocol that compares y to *many* values has the same number of random OTs as the basic comparison protocol comparing y to a *single* value, but it requires the transferred strings to be of length $n_1(\lambda + \log_2(n_1))$ bits instead of λ bits. Note, however, that since we use random OTs there is no need to send these strings in the OT protocol. Instead, all strings corresponding to the same value of the same input bit can be generated from a single seed using a pseudo-random generator. Therefore, the amount of data transferred in the OTs is the same as for the single comparison PEQT protocol.

The only additional data that is sent is the $n_1(\lambda + \log_2 n_1)$-bit string m_{P_1}, which P_1 sends to P_2. Hence, the total amount of communication is $2\sigma\kappa/\eta + n_1(\lambda + \log_2 n_1)$ bits.

In addition, the PRG which is used to generate the output string from the OT must be evaluated multiple times to generate the $n_1(\lambda + \log_2 n_1)$ bits. Therefore, the set inclusion protocol, which compares y to n_1 elements, is less efficient than a single run of the PEQT protocol, but is definitely more efficient than n_1 invocations of the PEQT protocol.

5.3 The OT-Based PSI Protocol

To obtain the final PSI protocol that computes $X \cap Y$, P_2 simply invokes the private set inclusion protocol of §5.2 for each $y \in Y$. Correctness and security follow from the properties of the private set inclusion protocol.

[3] Note that for $\sigma < \kappa$ we can perform σ base-OTs instead of using OT extension. However, here we analyze the costs when using OT extension for simplicity and consistency reasons.

Efficiency Overall, to compute the intersection between sets X and Y of σ-bit elements, the protocol requires $n_2\sigma/\eta$ random $\binom{N}{1}$-OTs of $n_1(\lambda+\log_2 n_1)$ bit-strings and additionally $n_1 n_2(\lambda+\log_2 n_1)$ bits to be sent. Using the random $\binom{N}{1}$-OT of [35], the total amount of communication is $2n_2\sigma\kappa/\eta + n_1 n_2(\lambda+\log_2 n_1)$ bits. For large n_1 and n_2, this amount of communication grows too large for an efficient solution. In order to cope with large sets, one can use a hashing scheme, as shown in §6.

6 Hashing Schemes and PSI

Several private set intersection protocols are based on running many invocations of pairwise private equality tests (PEQT). These protocols include [11, 21, 27] or our set inclusion protocol in §5. A straightforward implementation of these protocols requires n^2 invocations of PEQT for sets of size n, and therefore does not scale well. In [19, 21] it was proposed to use hashing schemes to reduce the number of comparisons that have to be computed. The idea is to have each party use a publicly known random hashing scheme to map its input elements to a set of bins. If an input element is in the intersection, both parties map it to the same bin. Therefore, the protocol can check for intersections only between items that were mapped to the same bin by both parties.

Naively, if n items are mapped to n bins then the average number of items in a bin is $O(1)$, checking for an intersection in a bin takes $O(1)$ work, and the total overhead is $O(n)$. However, privacy requires that the parties hide from each other how many of their inputs were mapped to each bin.[4] As a result, we must calculate in advance the number of items that will be mapped to the *most populated* bin (w.h.p.), and then set all bins to be of that size. (This can be done by storing dummy items in bins which are not fully occupied.) This change hides the bin sizes but also increases the overhead of the protocol, since the number of comparisons per bin now depends on the size of the most populated bin rather than on the actual number of items in the bin. However, while the parties need to pretend externally that all their items are real, they do not need to apply all their internal computations to their dummy items (since they know that these items are not in the intersection). A careful implementation of this observation, which takes into account timing attacks, can further optimize the computation complexity of the underlying protocols.

The work of [19, 21] gave asymptotic values for the bin sizes that are used with this technique, and of the resulting overhead. They left the task of setting appropriate parameters for the hashing schemes to future work. We revisit the hashing schemes that were outlined in [19, 21], namely, simple hashing, balanced allocations, and Cuckoo hashing (§6.1). We evaluate the performance when using hashing schemes for PSI (§6.2), and describe an analysis of the involved parameters (§6.3). We conclude that Cuckoo hashing yields the best performance (for parameters which we find to be most reasonable).

6.1 Hashing Schemes

Simple Hashing In the simplest hashing scheme the hash table consists of b bins $B_1 ... B_b$. Hashing is done by mapping each input element e to a bin $B_{h(e)}$ using a hash function $h : \{0,1\}^\sigma \mapsto [1,b]$ that was chosen uniformly at random and independently of the input elements. An element is always added to the bin to which it is mapped, regardless of whether other elements are already stored in that bin. Estimating the maximum number of elements that are mapped to any bin, denoted max_b, is a non-trivial problem and has been subject to extensive research [24, 38, 48]. When hashing m elements to $b = m$ bins, [24] showed that $max_b = \frac{\ln m}{\ln\ln m}(1+o(1))$ w.h.p. In this case, there is a difference between the expected and the maximum number of elements mapped to a bin, which are 1 and $O(\frac{\ln m}{\ln\ln m})$, respectively. When decreasing the number of bins to a value b satisfying $c \cdot b \ln b = m$ for some constant c, it was shown in [48] that $max_b = (d_c - 1 + \alpha)\ln b$, where d_c is the largest solution to $f(x) = 1 + x(\ln c - \ln x + 1) - c = 0$, and α is a parameter for adjusting the conservativeness of the approximation, and should be set to be slightly larger than 1. In this case the expected and maximum number of elements mapped to a bin are of the same order $O(\ln b) \approx O(\ln m)$. This is preferable for our purposes, since even though privacy requires that we set each bin to be as large as the most populated bin, this size is of the same order as the expected size of a bin when no privacy is needed.

Balanced Allocations The balanced allocations hashing scheme [2] uses two uniformly random hash functions $h_1, h_2 : \{0,1\}^\sigma \mapsto [1,m]$. An element e is mapped by checking which of the two bins $B_{h_1(e)}$ and $B_{h_2(e)}$ is less occupied, and mapping the element to that bin. A lookup for an element q is then performed by checking both bins, $B_{h_1(q)}$ and $B_{h_2(q)}$, and comparing the elements in these bins to q. The advantage of this scheme, shown in [2], is that when hashing m elements into $b = m$ bins, max_b is only $\frac{\ln\ln m}{\ln 2}(1+o(1))$, i.e., exponentially smaller than in simple hashing.

[4] Otherwise, and since the hash function is public, some information is leaked about the input. For example, if no items of P_1 were mapped to the first bin by the hash function h, then P_2 learns that P_1 has no inputs in the set $h^{-1}(1)$, which covers about $1/n$ of the input range.

Cuckoo Hashing Similar to balanced allocations hashing, Cuckoo hashing [46] uses two hash functions $h_1, h_2 : \{0,1\}^\sigma \mapsto [1,b]$ to map m elements to $b = 2(1+\varepsilon)m$ bins. The scheme avoids collisions by relocating elements when a collision is found using the following procedure: An element e is inserted into a bin $B_{h_1(e)}$. Any prior contents o of $B_{h_1(e)}$ are evicted to a new bin $B_{h_i(o)}$, using h_i to determine the new bin location, where $h_i(o) \neq h_1(e)$ for $i \in \{1,2\}$. The procedure is repeated until no more evictions are necessary, or until a threshold number of relocations been performed. In the latter case, the last element is put in a special stash s. It was shown that for a stash of size $s \leq \ln m$, insertion of m elements fails with probability m^{-s} [33]. A lookup in this scheme is very efficient as it only compares e to the two items in $B_{h_1(e)}$ and $B_{h_2(e)}$ and to the s items in the stash. In exchange for the improved lookup overhead, the size of the hash table is increased to about $2m$ bins.

6.2 Evaluation of Hashing-Based PSI

We evaluate the asymptotic overhead of applying the OT-based PSI protocol that was introduced in §5.3 while using any of the hashing scheme that we described. Also note that P_1 can save communication since instead of sending all masks for each bin (including masks for both dummy and real values), it can send only the masks of its real values (in permuted order, so that P_2 does not know which value was in each bin). P_2 can then simply check every mask received from P_1 against every computed mask. However, in this case the bit-length ℓ of the masks has to be increased to $\ell' = \lambda + \log_2 n_1 + \log_2 n_2$, since P_2 has to perform a total of $n_1 n_2$ comparisons and the overall error probability must be at most $2^{-\lambda}$. In the following, we address the mask length for checking one item against a set of n_1 items as $\ell_1 = \lambda + \log_2 n_1$ and the mask length for checking a set of n_2 items against n_1 items as $\ell_2 = \lambda + \log_2 n_1 + \log_2 n_2$.

PSI based on simple hashing A protocol based on simple hashing allocates the n inputs of P_2 to b bins, such that $n = O(b \ln b)$ and b is approximately $O(n/\ln n)$. Each bin is padded with dummy items to contain the maximum number of items that is expected in a bin, which is $O(\ln b) = O(\ln n)$. For each bin, the parties need to compute the intersection between sets of $O(\ln n)$ items. Each item can be represented using $O(\ln \ln n)$ bits.[5] The protocol requires $O(\ln n \ln \ln n)$ random OTs for each bin. The total number of OTs is therefore $O(n \ln \ln n)$. The length of the values transferred in the OTs (the masks) is $\ell_2 \ln n$ bits.

[5] This holds since the items in a bin can be hashed to a shorter representation, as long as no collisions occurs. The length of the hashed value should be about $\lambda + \log((\ln n)^2) = O(\ln \ln n)$.

PSI based on balanced allocations A major problem occurs when using balanced allocations hashing for PSI: every item can be mapped to one of two bins, and therefore it is unclear with which of P_1's bin should P_2 compare its own input elements e. Furthermore, the protocol must hide from each party the choice of bins made by the other party to store e, since that choice depends on other input elements and might reveal information about them. The solution to this is to use balanced allocations by P_2 alone, whereas P_1 maps each of its input elements to *two* bins using simple hashing with both hash functions h_1 and h_2. When using $b = n$ bins, P_2 has $O(\ln \ln n)$ items in each bin, whereas P_1 has $O(\ln n / \ln \ln n)$ items in every bin (actually, it has twice as many items as with simple hashing, since it maps each item twice). The items can be represented using strings of $O(\ln \ln n)$ bits. The protocol continues as before. P_2 learns the output, but since P_1 does not use balanced allocations, P_1 does not learn P_2's choices in that hashing scheme. The number of OTs is linear in the number of items stored by P_2 multiplied by the representation length, e.g., $O(n \cdot (\ln \ln n)^2)$ OTs on $\ell_2 \ln n / \ln \ln n$ bit strings. This overhead is larger than that of the simple hashing-based scheme.

PSI based on Cuckoo hashing Designing PSI based on Cuckoo hashing encounters the same privacy problem as when using balanced allocations hashing, and therefore the same solution is used. P_2 uses Cuckoo hashing whereas P_1 maps each of its elements using simple hashing with each of the two hash functions. P_2 maps a single item to each of the $2n$ bins, whereas P_1's bins contain $O(\ln n)$ items. In addition, P_2 has a stash of $s \leq \ln n$ elements. Each of these elements must be compared with each of P_1's n elements. An item in a bin can again be represented using $O(\ln \ln n)$ bits, whereas an item in the stash can be represented using $O(\ln n)$ bits. Furthermore, when checking items in the stash, we check one item against n_1, allowing us to reduce the bit-size of the masks in the OTs to ℓ_1 instead of ℓ_2. The protocol therefore performs $O(n \ln \ln n)$ OTs on inputs of length $O(\ell_2 \ln \ln n)$ bits (for the items in the bins), and in addition $O((\ln n)^2)$ OTs of inputs of length $O(\ell_1 \ln n)$ bits (for the items in the stash, which are each compared to all items of P_1's input). Overall, the protocol has the same asymptotic overhead as the protocol that uses simple hashing.

6.3 Maximum Bin Size and Overhead

When using hashing schemes for private set intersection, the number of bins b and the corresponding maximum bin size max_b must be set to values that balance efficiency and security. If max_b is chosen too small, the probability of a party failing to perform the mapping, denoted P_{fail}, increases. As a result, the output might be inaccurate

Parameter	Total # OTs	Comm. [bits] P_1 to P_2	Comm. [MB] total, $n = 2^{18}$
No hashing	nt	$n^2\ell_1$	458,880 / 458,784
Simple Hashing	$3.7nt$	$n\ell_2$	476 / 121
Balanced Alloc.	$2.9nt(\ln\ln n)$	$2n\ell_2$	1,298 / 595
Cuckoo Hashing	$(2(1+\varepsilon)n+s)t$	$sn\ell_1+2n\ell_2$	319 / 89

Table 4: Number of OTs and communication for the different hashing-based protocols. The total communication given in the last column is calculated for $\ell_1 = \lambda + \log_2 n$, $\ell_2 = \lambda + 2\log_2 n$, $\kappa = 128$, $\lambda = 40$, $\varepsilon = 0.2$, $s = 4$. The first value in this column is for $t = \sigma = 32$ $\binom{2}{1}$-OTs per element and the second value is for $t = 32/8$ $\binom{N}{1}$-OTs, $N = 2^8$. It is composed of the total number of OTs in the 2nd column times the communication per OT plus the communication from P_1 to P_2 in the 3rd column.

(since not all items can be mapped to bins), or one of the parties needs to request a new hash function (a request that leaks information about the input set of that party). On the other hand, the number of performed comparisons increases with b and max_b. An asymptotic analysis of the maximum bin size was presented in [19, 21], but leaves the exact choice of b and max_b and the resulting P_{fail} to further work. In the following, we analyze the complexity of the hashing schemes when used in combination with our set inclusion protocol, described in §5. To compare the performance of the hashing schemes on a unified base, we depict in Tab. 4 the overall communication, divided into the number of OTs (where we run t OTs per element) and the number of bits sent from P_1 to P_2.

In the full version [47]we detail the analysis of setting the optimal parameters for usage of the different hashing schemes in our PSI protocol, and of the resulting number of OTs and communication overhead. The results are depicted in Tab. 4 and show that Cuckoo hashing has the lowest communication. In addition, this scheme has a stronger guarantee on the upper bound of P_{fail}, since we achieve rehash probability of n^{-s}. We therefore use this scheme in our implementation and experiments.

A note on approximations When using a hashing scheme with fixed bin sizes it is possible that the number of items mapped to a certain bin, say by P_1, is larger than the capacity of the bin. (This event happens with probability P_{fail}.) In such a case it is possible for P_1 to ask to use a new hash function. This request reveals some information about P_1's input. Another option is for P_1 to ignore the missed item, and therefore essentially compute an approximation to the intersection. This choice, too, might reveal information about P_1's input, albeit in a more subtle way through multiple invocations of the functionality. Similarly, in the Bloom filter-based protocol, the occurrence of a false positive might leak information. The best solution to this issue is to make sure that the probability of these events happening is negligible, so that it is almost certain that these events will not occur in practice. This is the approach that we take in our comparisons. (Another approach would be to allow the computation of an approximation of the original intersection function, while analyzing the privacy leakage effects of this computation, and deciding whether to tolerate them. The result might be a more liberal choice of parameters which will result in a more efficient implementation of the original protocol.)

7 Experimental Evaluation

In the following we experimentally evaluate the PSI protocols described before. We describe our benchmarking environment in §7.1 and then detail the comparison between the protocols in §7.2. Tab. 5 compares the single-threaded runtimes of all protocols over Gigabit LAN, Tab. 6 compares the communication complexities, and Tab. 7 compares the single-threaded runtimes on different networks. In the tables we highlight the protocol with lowest runtime and communication for each type.

7.1 Benchmarking Environment

We ran our experiments on two Intel Core2Quad desktop PCs (**without** AES-NI extension) with 4 GB RAM, connected via Gigabit LAN. In each experiment, P_1 and P_2 held the same number of input elements n and were not allowed to perform any pre-computation. We set n as in [17], i.e., $n \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$, but omitted $n = 2^{20}$, since many implementations exceeded the available main memory. We use $\sigma = 32$ as the bit length of the elements.[6] We use a statistical security parameter $\lambda = 40$ and a symmetric security parameter $\kappa \in \{80, 128\}$ (other security parameters are chosen according to Tab. 2). For our set-inclusion protocol we set $\eta = 8$, i.e., use 1-out-of-2^8 OT extensions.

In our tables, the asymptotic performance is given for the party with the majority of the workload, and are divided to public-key operations (asym) and symmetric cryptographic operations (sym).

Implementations The implementation of the blind-RSA-based [14] and garbled Bloom-Filter [17] protocols were taken from the authors, but we used a hash-table to compute the last step in the blind-RSA protocol that finds the intersection (the original implemen-

[6] For protocols whose complexity depends on σ, elements from a large domain can be hashed to short representations; cf. full version [47] for details.

tation used pairwise comparisons with quadratic runtime overhead). We implemented a state-of-the-art Yao's garbled circuits protocol (using garbled-row-reduction, point-and-permute, free-XOR, and pipelining, cf [28]) by building on the C++ implementation of [12] and using the fixed-key garbling scheme of [7][7]. For Yao's garbled circuits protocol, we evaluated a size-optimized version of the sort-compare-shuffle circuit (comparison circuits of size and depth σ) while for GMW we evaluated a depth-optimized version (comparison circuits of size 3σ and depth $\log_2 \sigma$) for σ-bit input values [49].

We implemented FFC (finite field cryptography) and IFC (integer factorization cryptography) using the GMP library (v. 5.1.2), ECC using the Miracl library (v. 5.6.1), symmetric cryptographic primitives using OpenSSL (v. 1.0.1e), and used the OT extension implementation of [1] which requires about 3 symmetric cryptographic operations per OT for the asymptotic performance analysis.

We argue that we provide a fair comparison, since all protocols are implemented in the same programming language (C/C++), run on the same hardware, and use the same underlying libraries for cryptographic operations.

For each protocol we measured the time from starting the program until the client outputs the intersecting elements. All runtimes are averaged over 10 executions.

7.2 Performance Comparison

We divide the performance comparison into three categories, depending on whether the protocol is based on public-key operations, circuits, or OT. Afterwards, we provide experiments for different networks and give a comparison between the best protocols in each category.

Public-Key-Based PSI For the public-key-based PSI protocols, we observe that the DH-based protocol of [29] outperforms the RSA-based protocol of [14] when using finite field cryptography (FFC). Similarly to [1], we also obtain the somewhat surprising result that for 80-bit security elliptic curve cryptography (ECC) using the Miracl library is slower than FFC using the GMP library. For larger security parameters, however, ECC becomes more efficient and outperforms FFC by a factor of 3 for 128-bit security for the DH-based protocol. (The reason for this phenomenon might be better implementation optimizations in the GMP library.) The advantage of the ECC-based protocol is its communication complexity, which is lowest among all PSI protocols, cf. Tab. 6. We note that a major advantage of these protocols is their simplicity, which makes them comparably easy to implement.

[7]The security of the fixed-key garbling scheme is somewhat controversial but we included it for performance reasons.

Circuit-Based PSI Here we tested Yao- and GMW-based implementations, as well as an implementation of our optimized vector multiplication-triple-based GMW protocol (§3.2). Following is a summary of the results:

- Both the computation complexity and the communication complexity of the circuit-based PSI protocols are the highest among all protocols that we tested.
- The basic GMW protocol has the highest overall runtime and communication complexity.
- Our vector multiplication triple optimization reduces the runtime and communication of GMW by approximately 40%. For security parameter $\kappa = 80$, this implementation is slightly faster than Yao's protocol, but it is slightly slower for $\kappa = 128$. Communication-wise, the vector multiplication triple GMW is more efficient than Yao's protocol.
- The runtime of Yao's protocol hardly increases with the security parameter, since we use AES-128 for both versions. Note, however, that our implementation of Yao's protocol exceeded the main memory when processing 2^{18} elements.
- Our Yao implementation does not use the AES-NI hardware support. Using AES-NI is likely to improve the runtime of the Yao implementation.

We give a more detailed performance comparison for GMW and Yao's protocol in the full version [47].

OT-Based PSI The random garbled Bloom filter protocol of §4.3 improves the original garbled Bloom filter protocol of [17] by more than a factor of two in runtime and by factor of 2-3 in communication.

We also implemented our protocol of §5, where we used Cuckoo hashing with parameters $\varepsilon = 0.2$ and $s = 4$, cf. §6. This protocol had the best runtime, and was about 5 times faster than the random garbled Bloom filter protocol for $\kappa = 128$. In terms of communication, our set inclusion protocol uses less than 20% of the communication of the random garbled Bloom filter protocol for $\kappa = 80$ and less than 10% communication for $\kappa = 128$.

The main difference between the set inclusion protocol and the random garbled Bloom filter protocol is the dependency of the performance on the symmetric security parameter κ. In the random garbled Bloom filter protocol, the number of OTs is independent of the bit-length σ but scales linearly with κ. On the other hand, the number of OTs for the set inclusion protocol is independent of κ but linear in σ. As a result, the runtime of the Bloom filter protocol (but not of the set inclusion protocol) is greatly affected when κ is increased.

Type	Symm. Security Parameter κ	80-bit					128-bit					Asymptotic
	Set Size n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	
Public-Key	**DH-based FFC [29]**	**0.4**	**1.6**	**6.2**	**24.7**	**98.8**	4.8	19.1	76.5	306.0	1,224.1	$2n$ asym
	DH-based ECC [29]	0.7	2.8	11.0	44.1	177.5	**1.6**	**6.5**	**26.1**	**104.2**	**416.2**	$2n$ asym
	RSA-based [14]	0.5	2.0	7.9	31.3	124.9	7.7	31.0	124.3	497.2	1,982.1	$2n$ asym
Circuit [27]	**Yao [7, 28]**	1.2	5.7	27.7	128.2	-	**1.6**	**6.3**	**28.4**	**129.1**	-	$12n\sigma\log_2 n$ sym
	GMW [1]	1.9	8.6	35.2	161.9	806.5	2.6	12.8	58.9	276.4	1,304.2	$30n\sigma\log_2 n$ sym
	Vector-MT GMW §3.2	**1.2**	**5.1**	**21.2**	**100.3**	**462.7**	1.9	7.8	36.5	168.9	762.4	$18n\sigma\log_2 n$ sym
OT	Garbled Bloom Filter [17]	0.3	0.9	3.9	16.1	71.9	0.6	2.0	8.5	37.1	154.4	$4.32n\kappa$ sym
	Random Garbled Bloom Filter §4.3	0.15	0.5	2.0	8.1	34.3	0.27	1.0	4.1	16.7	67.6	$3.6n\kappa$ sym
	Set Inclusion §5 + Hashing §6	**0.13**	**0.2**	**0.8**	**3.3**	**13.5**	**0.26**	**0.3**	**0.9**	**3.7**	**13.8**	$0.75n\sigma$ sym

Table 5: Runtimes in seconds for PSI protocols with one thread over Gigabit LAN ($\sigma = 32$: bit size of set elements, asym: public-key operations, sym: symmetric cryptographic operations).

Type	Symm. Security Parameter κ	80-bit					128-bit					Asymptotic
	Set Size n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	
Public-Key	DH-based FFC [29]	0.4	1.5	6.0	24.0	96.0	1.1	4.5	18.0	72.0	288.0	$3n\rho$
	DH-based ECC [29]	**0.1**	**0.2**	**1.0**	**3.8**	**15.0**	**0.1**	**0.4**	**1.5**	**6.0**	**24.0**	$3n\varphi$
	RSA-based [14]	0.3	1.1	4.3	17.3	69.0	0.8	3.1	12.5	50.0	200.0	$2n\rho + 2n\kappa$
Circuit [27]	Yao [7, 28]	28.1	135.0	630.0	2,880.0	12,960.0	45.0	216.0	1,008.0	4,608.0	20,736.0	$9n\kappa\sigma\log_2 n$
	GMW [1]	31.3	150.0	700.0	3,200.0	14,400.0	50.0	240.0	1,120.0	5,120.0	23,040.0	$10n\kappa\sigma\log_2 n$
	Vector-MT GMW §3.2	**18.8**	**90.0**	**420.0**	**1,920.0**	**8,640.0**	**30.0**	**144.0**	**672.0**	**3,072.0**	**13,824.0**	$6n\kappa\sigma\log_2 n$
OT	Garbled Bloom Filter [17]	3.4	13.5	54.0	216.0	864.0	7.6	30.2	121.0	483.8	1,935.4	$2.88n\kappa(\kappa+\lambda)$
	Random GBF §4.3	1.1	4.5	18.1	72.6	290.4	2.9	11.6	46.2	184.9	739.7	$1.44n\kappa^2 + n(\lambda + 2\log_2 n)$
	Set Inclusion §5 + Hashing §6	**0.2**	**0.8**	**3.3**	**13.4**	**54.3**	**0.3**	**1.2**	**4.8**	**19.4**	**78.3**	$0.5n\kappa\sigma + 6n(\lambda + 2\log_2 n)$

Table 6: Communication complexity in MB for PSI protocols. ($\sigma = 32$: bit size of set elements, security parameters $\kappa, \lambda, \rho, \varphi$ as defined in §2.1). Numbers are computed from the asymptotic complexity given in the last column.

Experiments for Different Networks For each protocol type (public-key-based, circuit-based, and OT-based), we benchmark the best performing PSI protocol in different network scenarios: Gigabit LAN, 802.11g WiFi, intra-country WAN, inter-country WAN, and mobile Internet (HSDPA) and depict our results in Tab. 7. We characterize each network scenario by its bandwidth and latency. By latency we mean one-way latency, i.e., the time from source to sink, and we used the same bandwidth for up- and downlink. We simulated these network types using the Linux command `tc` and ran the protocols on $n = 2^{16}$ elements for $\kappa = 128$ and with one thread.

The only protocol that is nearly unaffected by the change in network environment and for which the network has not become the bottleneck is the DH-based ECC protocol. In this protocol computation is the bottleneck which can be improved by using multiple threads.

For the other protocols we observe how the main bottleneck transitions from computation to communication:

For Yao's protocol this transition happens very early, already when changing from Gigabit LAN to WiFi (factor 6 in runtime).[8] Our vector-MT GMW protocol and our random garbled Bloom filter protocol suffer less drastically from the decreased bandwidth (factor 2.3 in runtime). However, from the WiFi connection on, the performance of all three protocol decreases approximately linear in the bandwidth. Note that, although our vector-MT GMW protocol has only 66% of the communication complexity of Yao's protocol, it is more than two times faster in slower networks. This can be explained by the direction of the communication. In Yao's protocol, the large garbled circuit is sent in one direction, whereas the communication in GMW can be evenly distributed in both directions s.t. it uses both up- and downlink.

For our set inclusion protocol, the network saturation happens when using intra-country WAN. From this point on, the performance also decreases linearly with the bandwidth. Still, this protocol is the fastest of all protocols in all network settings.

Experiments with Multiple Threads Tab. 8 shows the runtimes with four threads. Of special interest is the last column, which shows the ratio between the runtimes with four threads and a single thread for $n = 2^{18}$ elements and security parameter $\kappa = 128$. The DH-based protocol, which is very simple and easily parallelizable, achieves almost the optimal speedup of 4x as computation is the performance bottleneck. The GMW protocol achieves only a speedup of about 2x, possibly due to the gate-by-gate evaluation of the circuit resulting in multiple rounds of communication as the bottleneck. The OT-based protocols achieve a very good speedup of about 3x.

[8] The performance advantage of using fixed-key AES garbling instead of SHA-1/SHA-256 already diminished in the WiFi setting.

Type	Network (Bandwidth (Mbit/s) / Latency (ms))	Gigabit LAN (1,000 / 0.2)	802.11g WiFi (54 / 0.2)	Intra-country WAN (25 / 10)	Inter-country WAN (10 / 50)	HSDPA (3.6 / 500)
Public-Key	DH-based ECC [29]	104.2	104.8	107.6	111.8	115.9
Circuit [27]	Yao [7, 28]	129.1	779.5	1,735.5	4,631.8	11,658.6
	Vector-MT GMW §3.2	168.9 (11.3)	370.5 (18.1)	770.4 (27.5)	1,936.5 (67.2)	5,310.9 (170.2)
OT	Random Garbled Bloom Filter §4.3	16.6	37.2	70.8	164.9	445.0
	Set Inclusion §5 + Hashing §6	**3.7**	**5.0**	**8.8**	**22.8**	**77.5**

Table 7: Runtimes in seconds for PSI protocols with one thread in different network scenarios for $n = 2^{16}$ elements, $\sigma = 32$: bit size of elements, and $\kappa = 128$-bit security (cf. Tab. 2); online time for Vector-MT GMW in ().

Type	Symm. Security Parameter κ	80-bit					128-bit					Speedup
	Set Size n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	
Public-Key	DH-based FFC [29]	0.1	0.5	1.8	6.7	26.9	1.3	5.2	20.8	80.1	320.1	**3.82x**
Circuit [27]	Vector-MT GMW §3.2	0.8	3.6	15.9	71.3	288.1	1.1	5.6	23.4	96.1	400.9	1.90x
OT	Random Garbled Bloom Filter §4.3	0.08	0.2	0.8	3.2	13.1	0.14	0.5	1.7	6.4	25.9	2.61x
	Set Inclusion §5 + Hashing §6	**0.04**	**0.16**	**0.4**	**1.2**	**4.7**	**0.04**	**0.2**	**0.5**	**1.4**	**4.9**	2.81x

Table 8: Runtimes in seconds for PSI protocols with four threads and $\sigma = 32$; speedup for $n = 2^{18}$ and $\kappa = 128$.

Comparison From the results we observe that OT-based protocols have the lowest runtime on a fast network. The public-key-based protocols require costly public-key operations, which scale very poorly with increasing security parameter, but need less communication than the OT- or circuit-based protocols. The circuit-based protocols have a smaller runtime than the public-key-based protocols using FFC or RSA for $\kappa = 128$, but by far the highest communication complexity.

Our set inclusion protocol achieves both the most efficient runtime and a very low communication overhead. Compared to the second fastest protocol, namely our optimized random garbled Bloom-filter protocol, the set inclusion protocol is at least 5 times faster and uses 10 times less communication (for 128 bit security). Moreover, this protocol has the second best communication overhead, requiring only 3 times the communication of the DH-ECC-based protocol of [29], but running faster in all network environments that we tested.

We stress that the choice of the preferable PSI protocol depends on the application scenario. For instance,

- If communication is the bottleneck and computation is vast, then the DH-based PSI protocol using ECC is the most favorable. That protocol is also the simplest protocol to implement.
- The circuit-based protocols are unique in that they are based on generic secure computation techniques and can therefore be easily modified to compute more complex variants of PSI.
- While our set inclusion protocol performs very efficiently for $\sigma = 32$, it would require twice the runtime for $\sigma = 64$, while the random garbled Bloom filter protocol would have approximately the same runtime (which would still be greater).

Acknowledgements We thank the anonymous reviewers of USENIX Security 2014 for their helpful comments on our paper. This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE), by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by a grant from the Israel Ministry of Science and Technology (grant 3-9094).

References

[1] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Computer and Communications Security (CCS'13)*, pages 535–548. ACM, 2013.

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing*, 29(1):180–200, 1999.

[3] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Computer and Communications Security (CCS'11)*, pages 691–702. ACM, 2011.

[4] R. W. Baldwin and W. C. Gramlich. Cryptographic protocol for trustable matchmaking. In *IEEE S&P'85*, pages 92–100. IEEE, 1985.

[5] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.

[6] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Symposium on Theory of Computing (STOC'96)*, pages 479–488. ACM, 1996.

[7] M. Bellare, V. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P'13*, pages 478–492. IEEE, 2013.

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Computer and Communications Security (CCS'93)*, pages 62–73. ACM, 1993.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. OpenConflict: Preventing real time map hacks in online games. In *IEEE S&P'11*, pages 506–520. IEEE, 2011.

[11] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: Custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN)*, 2013.

[12] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in online marketplaces. In *Cryptographers' Track at the RSA Conference (CT-RSA'12)*, volume 7178 of *LNCS*, pages 416–432. Springer, 2012.

[13] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *Advances in Cryptology – ASIACRYPT'10*, volume 6477 of *LNCS*, pages 213–231. Springer, 2010.

[14] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, pages 143–159. Springer, 2010.

[15] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing (TRUST'12)*, volume 7344, pages 55–73. LNCS, 2012.

[16] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 125–142. Springer, 2009.

[17] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. In *Computer and Communications Security (CCS'13)*, pages 789–800. ACM, 2013.

[18] M. Fischlin, B. Pinkas, A.-R. Sadeghi, T. Schneider, and I. Visconti. Secure set intersection with untrusted hardware tokens. In *Cryptographers' Track at the RSA Conference (CT-RSA'11)*, volume 6558 of *LNCS*, pages 1–16. Springer, 2011.

[19] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set-intersection with simulation-based security. In *Journal of Cryptology*, 2013. To appear.

[20] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference (TCC'05)*, volume 3378 of *LNCS*, pages 303–324. Springer, 2005.

[21] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology – EUROCRYPT'04*, volume 3027 of *LNCS*, pages 1–19. Springer, 2004.

[22] O. Goldreich. *Foundations of Cryptography*, volume 2: Basic Applications. Cambridge University Press, 2004.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Symposium on Theory of Computing (STOC'87)*, pages 218–229. ACM, 1987.

[24] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

[25] C. Hazay and Y. Lindell. Constructions of truly practical secure protocols using standardsmartcards. In *Computer and Communications Security (CCS'08)*, pages 491–500. ACM, 2008.

[26] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Public Key Cryptography (PKC'10)*, volume 6056 of *LNCS*, pages 312–331. Springer, 2010.

[27] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed System Security (NDSS'12)*. The Internet Society, 2012.

[28] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, pages 539–554. USENIX, 2011.

[29] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *ACM Conference on Electronic Commerce (EC'99)*, pages 78–86. ACM, 1999.

[30] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[31] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *Theory of Cryptography Conference (TCC'09)*, volume 5444 of *LNCS*, pages 577–594. Springer, 2009.

[32] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. In *Financial Cryptography and Data Security (FC'14)*, LNCS. Springer, 2014.

[33] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

[34] L. Kissner and D. Song. Privacy-preserving set operations. In *Advances in Cryptology – CRYPTO'05*, volume 3621 of *LNCS*, pages 241–257. Springer, 2005.

[35] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *Advances in Cryptology – CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.

[36] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *IEEE S&P'86*, pages 134–137. IEEE, 1986.

[37] G. Mezzour, A. Perrig, V. D. Gligor, and P. Papadimitratos. Privacy-preserving relationship path discovery in social networks. In *Cryptology and Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 189–208. Springer, 2009.

[38] M. D. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[39] P. Mohassel and S. S. Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *Advances in Cryptology – EUROCRYPT'13*, volume 7881 of *LNCS*, pages 557–574. Springer, 2013.

[40] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *USENIX Security Symposium*, pages 95–110. USENIX, 2010.

[41] M. Nagy, E. De Cristofaro, A. Dmitrienko, N. Asokan, and A.-R. Sadeghi. Do I know you? – efficient and privacy-preserving common friend-finder protocols and applications. In *Annual Computer Security Applications Conference (ACSAC'13)*, pages 159–168. ACM, 2013.

[42] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SIAM Symposium On Discrete Algorithms (SODA'01)*, pages 448–457. Society for Industrial and Applied Mathematics (SIAM), 2001.

[43] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Network and Distributed System Security (NDSS'11)*. The Internet Society, 2011.

[44] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO'12*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[45] NIST. NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Rev. 3). Technical report, National Institute of Standards and Technology (NIST), 2012.

[46] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA'01)*, volume 2161 of *LNCS*, pages 121–133. Springer, 2001.

[47] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. Cryptology ePrint Archive, Report 2014/447, 2014. `http://eprint.iacr.org/2014/447`.

[48] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science (RANDOM'98)*, volume 1518 of *LNCS*, pages 159–170. Springer, 1998.

[49] T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security (FC'13)*, volume 7859 of *LNCS*, pages 275–292. Springer, 2013.

Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data

Sebastian Vogl*, Robert Gawlik†, Behrad Garmany†, Thomas Kittel*,
Jonas Pfoh*, Claudia Eckert*, Thorsten Holz†

*Technische Universität München
†Horst Görtz Institute for IT-Security, Ruhr-University Bochum

Abstract

Generally speaking, malicious code leverages *hooks* within a system to divert the control flow. Without them, an attacker is blind to the events occurring in the system, rendering her unable to perform malicious activities (e.g., hiding of files or capturing of keystrokes). However, while hooks are an integral part of modern attacks, they are at the same time one of their biggest weaknesses: Even the most sophisticated attack can be easily identified if one of its hooks is found. In spite of this fact, hooking mechanisms have remained almost unchanged over the last years and still rely on the *persistent* modification of *code* or *control data* to divert the control flow. As a consequence, hooks represent an abnormality within the system that is *permanently evident* and can in many cases easily be detected as the hook detection mechanisms of recent years amply demonstrated.

In this paper, we propose a novel hooking concept that we refer to as *dynamic hooking*. Instead of modifying persistent control data permanently, this hooking mechanisms targets *transient* control data such as return addresses at run-time. The hook itself will thereby reside within *non-control data* and remains hidden until it is triggered. As a result, there is no *evident* connection between the hook and the actual control flow change, which enables dynamic hooks to successfully evade existing detection mechanisms. To realize this idea, dynamic hooks make use of exploitation techniques to trigger vulnerabilities at run-time. Due to this approach, dynamic hooks cannot only be used to arbitrarily modify the control flow, but can also be applied to conduct *non-control data* attacks, which makes them more powerful than their predecessors. We implemented a prototype that makes uses of static program slicing and symbolic execution to automatically extract paths for dynamic hooks that can then be used by a human expert for their realization. To demonstrate this, we used the output provided by our prototype to implement concrete examples of dynamic hooks for both modern Linux and Windows kernels.

1 Introduction

Over the last decade, the sophistication and technical level of malicious software (*malware*) has increased dramatically. In the early 2000s, we saw malware such as the *I Love You* [29] and *Blaster* worms [3] that generally operated in user space with very little in the way of defensive mechanisms. In contrast, we nowadays see complex kernel level malware such as *Stuxnet*, *Duqu*, and *Flame* [6] that show an increase in sophistication in the target of their attack, the exploitation methods used to deliver them, and their ability to evade detection. By targeting kernel space, modern malware effectively runs at the same privilege level as the operating system (OS), enabling it to attack and modify any part of the system including the kernel itself. In addition, malware can take advantage of stealth techniques that were originally only used by kernel rootkits to hide itself deep within the system. This makes the detection of malware increasingly difficult, especially as malware continues to evolve, enabling it to stay one step ahead of the defenders. Currently, one the most sophisticated methods employed is data-only malware [16, 41].

However, even very sophisticated malware such as data-only malware has an Achilles' heel: in general, malware needs to intercept events within the system to be able to fulfill its purpose [27, 44]. Without this capability, malware would be unable to react to events or provide fundamental functionality such as key logging and file hiding, which would severely limit its possibilities. Event interception, however, requires malware to divert the control flow of the infected system at run-time. To achieve this, malware must install *hooks* in the system that facilitate the required control flow transfer on behalf of the malware whenever the desired event occurs. While malware might manage to hide itself, these hooks represent an abnormality that will be *permanently* visible within the system and thus lends itself well to becoming the basis of detection mechanisms. This insight led to a wide range of research that enable the monitoring of malware hooking behavior for the purpose of signature

generation [47] or detecting malware based on control flow modifications [15, 20, 43].

Although existing systems are not yet able to detect all hooks that are placed by malware, the remaining possibilities for malware to install hooks are constantly dwindling. Hooks that are based on code modifications are usually no longer an option, since changes to code areas can be easily detected due to their static nature. This leaves attackers only with the option of data hooks [43, 47], but even here the options are increasingly restricted by modern detection mechanisms. The reason for this development is that, in contrast to malware where one can observe a constant evolution of techniques and mechanisms used, hooking techniques have not significantly changed over the course of recent years.

In this paper, we present a novel hooking concept that we refer to as *dynamic hooking*. In contrast to existing hooking mechanisms which *persistently* modify control data, our hooking approach targets *transient* control data such as return addresses at run-time. As a consequence, the resulting control flow change that is introduced by the hook only becomes visible when the hook is actually triggered. This significantly complicates the detection of dynamic hooks as security mechanisms can no longer focus on persistent control data, but must also take transient control data into account. What is even more, the hook itself will thereby reside in *non-control* data, which is much more difficult to analyze and verify [4, 15] when compared to *control* data that traditional hooks target. Despite the fact that dynamic hooks reside purely in non-control data, they are able to reliably intercept the execution flow of functions similar to traditional hooks. Furthermore, they can be used in pure data-only attacks, which are by themselves a realistic and dangerous threat [3, 5]. Thus they are not only harder to detect, but also more powerful than their predecessors.

To provide these capabilities, dynamic hooks modify data in such a way that they will trigger vulnerabilities at run-time. Through this approach, they are able to arbitrarily modify the control flow, while the hook itself only consists of the data that triggers and exploits the vulnerability. This makes them quite similar to traditional exploitation techniques with the exception that they target applications that are *already* controlled by the attacker. Due to this fact, the attack surface for dynamic hooks is much broader compared to traditional exploitation, since an attacker can not only attack external functions, but also *internal* functions.

Furthermore, dynamic hooks can be obtained automatically in a manner comparable to automated exploit generation [2]. To demonstrate this, we implemented a prototype that leverages static program slicing [40, 45] and symbolic execution [34] to automatically extract satisfiable, exploitable paths for dynamic hooks. The prototype thereby provides detailed information about each jump condition in the path and the actual vulnerability in an intuitive format, which makes the output suitable for exploit generation frameworks or a human expert. We used this prototype to automatically identify dynamic hooks for recent Linux and Windows kernels. Additionally, we implemented proof of concepts (POCs) of dynamic hooks that demonstrate how they can be used in practice to intercept events such as system calls or to implement backdoors. This proves that the suggested hooking mechanism is not only powerful, but also realistic.

In summary, we make the following contributions:

- We present a novel hooking concept called *dynamic hooking* that targets transient control data at runtime instead of persistent control-data. This approach bypasses existing hook detection techniques proposed in the last few years.
- We show how dynamic hooks for OS kernels can be automatically found by leveraging binary analysis techniques and implemented a prototype.
- We provide detailed POC implementations of dynamic hooks for both Linux and Windows kernels that demonstrate their capabilities and possibilities.

2 Technical Background

Before presenting our approach to realize dynamic hooks, we first review background information that is essential for the understanding of the remainder of the paper. We begin by defining important terms and then discuss why malware in general requires hooks within the system to function. Finally, we cover existing hooking mechanisms and their countermeasures.

2.1 Definitions

We first introduce important terms that we will use throughout the paper. In particular, we highlight the differences between control data and non-control data as well as transient and permanent control data.

Control data and non-control data. *Control data* specifies the target location of a branch instruction. By changing control data, an attacker can arbitrarily change the control flow of an application. Examples of control data are return addresses and function pointers.

In contrast, *non-control data* never contains the target address for a control transfer. In certain cases, however, it may influence the control flow of an application. For instance, a conditional branch may depend on the value of non-control data.

Transient and persistent control data. We consider control data to be *transient* when it cannot be reached through a pointer-chain originating from a global variable. This essentially implies that there is no lasting connection between the application and the control data. Instead, the control data is only visible in the current scope of the execution such as a return address which is only valid as long as a function executes.

By extension, we consider all control data that is reachable through a global variable as *persistent*, since the control data is permanently connected to the application and can thus always be accessed independent of the current scope.

2.2 Malware and Hooking

Petroni et al. [27] estimated that about 96% of all rootkits require *hooks* within the system to function. Intuitively, this makes sense: since the sole purpose of rootkits is to provide stealth, they have to hide all signs of an infection. While existing structures can be hidden using techniques such as *direct kernel object manipulation (DKOM)* [38], hooks enable rootkits to react to changes occurring at *run-time*. Consider, for instance, that a hidden process creates a new network connection or a child process. Naturally, a rootkit must also hide such newly created objects to achieve its goal. This, however, requires a rootkit to be notified of the occurrence of such events. Hooks solve this problem by enabling a rootkit to install callback functions in the system. This makes them an integral part of rootkit functionality.

In practice, rootkit functionality is often mixed with a variety of malicious payloads. According to a report by Microsoft released in 2012 [13], "some of the most prevalent malware families today consistently use rootkit functionality". The primary reason for this is that the single purpose of a rootkit is to avoid detection. Consequently, it is not a big surprise that the techniques formerly only found in rootkits are increasingly being adapted by malware. Since rootkits require hooks to function, this, however, also implies that any malware based on rootkit functionality will require the same.

2.3 Existing Hooking Mechanisms

In general, we distinguish between two different types of hooks: *code* hooks and *data* hooks [11, 43, 47]. Code hooks work by directly patching the application's code regions: wherever the attacker wants to redirect the control flow of the application, she overwrites existing instructions with a branch instruction. As a result, the control flow of the application is diverted every time the execution passes through the modified instructions.

The main problem with code hooks is that code regions are usually static. Therefore, it is generally sufficient to identify modifications to code regions to detect this type of hook. Various techniques have been proposed that leverage such an approach [22, 27, 31]. As a result, adversaries resorted to a different hooking form referred to as data hooks. Instead of modifying code directly, data hooks target *persistent* control data within the application. By modifying control data, the attacker is able to divert every control transfer that makes use of the modified data. For example, the most straightforward method for intercepting the execution of system calls is to modify function pointers within the system call table.

To counter the threat of data hooks, researchers proposed various systems that aim to protect control data within an application [9, 20, 27, 43]. However, the main focus of these systems thereby lies in the protection of function pointers that are allocated on the heap or reside within the data region of the application. This is achieved by ensuring that each function pointer points to a valid function according to its control flow graph (CFG). *Transient* control data on the other side is generally ignored by these approaches or they merely consider the protection of return addresses, which is not the only kind of transient control data. Instead transient function pointer may also exist as we will discuss in Section 5.1.

While researchers acknowledge that malware could potentially also target transient control data to modify the control flow [20, 27, 43], these attacks are usually only considered in the context of exploitation or return-oriented rootkits [16], but are not deemed to be relevant for hooking. The reasoning behind this assumption is that malware generally wants to change the control flow of the target application indefinitely in order to be continuously able to intercept events. Consequently, the malware must *permanently* redirect the control flow and thus target persistent control data as transient control data is, by definition, only used by the system for a limited amount of time. In this paper, we demonstrate that this assumption is false and can be used to circumvent existing defense mechanisms against hooking.

3 Dynamic Hooks

In the following, we introduce our novel hooking concept that we refer to as *dynamic hooking*. For this purpose, we provide an overview of the concept, discuss the vulnerabilities that can be used to implement dynamic hooks, and cover the types of dynamic hooks that exist and their properties. Before doing so, however, we first state the attacker model that we assume throughout this paper.

3.1 Attacker Model & Scope

In the following, we assume that the attacker's goal is to install persistent kernel malware such as a rootkit on the victim's system. For this purpose, we assume that the attacker has the ability to manipulate the kernel's memory arbitrarily either through a vulnerability or the ability to load a kernel module (or driver). To avoid detection, the attacker wishes to hide all the hooks of the malware. That is, we are not concerned with the stealth of the malware itself, but instead solely focus on its hooks. Consequently, we consider all malware detection mechanisms that do not detect malware based on its hooks to be out of scope for the remainder of the paper. Furthermore, we assume the target system leverages common protection mechanisms such as Address Space Layout Randomization (ASLR), stack canaries, and $W \oplus X$.

3.2 High-Level Overview

The main problem with existing hooking mechanisms is that they require the *permanent* change of code or function pointers. Consequently, the desired control flow change of the malware is *permanently* evident within the system [27]. The fundamental idea behind dynamic hooks is to solve this problem by hiding the desired control flow change within *non-control data* such that there is no clear connection between the changes that the malware conducts and the actual control flow change. This is accomplished with the help of *exploitation techniques*.

To exploit a vulnerable application, an attacker makes use of specially crafted input data that—when processed by the application—will eventually trigger a vulnerability. If the vulnerability enables the attacker to overwrite important control structures such as a return address, she will be able to modify and often control the execution flow of the application using techniques such as return-oriented programming (ROP) [35].

With dynamic hooks, we apply the same concepts that are used in traditional exploitation scenarios to hooking. That is, we manipulate the input data of the functions we want to hook in such a way that we will trigger a control flow modifying vulnerability when the data is used. This effectively allows us to overwrite control data (e.g., a return address) at run-time and enables us to control the execution flow of the application similar to a traditional hook. The main difference, however, is that such a dynamic hook will reside somewhere within the data structures of the application unnoticed until its malicious payload is eventually used by the target function.

For this approach to work, we need to identify a control flow modifying vulnerability in every function that we want to hook. At first glance this seems unlikely. However, there is a key difference between the exploitation of traditional vulnerabilities and vulnerabilities that are used to realize dynamic hooks: the attacker *already* controls the application at the time she installs a hook. In a traditional exploit, the attacker's goal is to *gain* control over an application. To achieve this, she needs to find an input to the application that will trigger a vulnerability. That is, the attacker can only control the *external* data which is provided to the application. In the case of a dynamic hook, however, this restriction does not apply. As the attacker controls the application, she is free to access and modify *any* internal data structure of the application. This results in a much stronger attacker model when compared to traditional exploitation.

Finding and exploiting vulnerabilities in such a scenario becomes much easier for several reasons. First, many existing protection mechanisms such as ASLR, stack canaries, or $W \oplus X$ only protect against an *external* attacker, but can be easily circumvented by an attacker that controls the application. Second, the attacker can *prepare* the code (or ROP chain) she wants to execute when the vulnerability is triggered beforehand and does not have to provide it during the exploitation process. This enables the attacker to exploit vulnerabilities for which traditional methods would be difficult due to space constraints of the vulnerability. Third, the attack surface for dynamic hooks is much broader. The attacker cannot only attack functions that handle user input, but can also target internal functions that cannot be influenced by the user. In fact, by manipulating internal data structures, the attacker can create new vulnerabilities that would not occur during normal operation of the application, because the targeted data structures are normally only accessed and modified by the program itself. This may even allow the attacker to circumvent checks and filters within the application as the manipulated data structures may contain values that could never occur during normal operation and may thus not have been expected by the programmer. Finally, to hook a specific event, the hook may be placed anywhere within the control flow of the handling code, it is not restricted to a single function.

Example. To illustrate the concept of dynamic hooks at a concrete example, consider the following code from the `list_del` function in the Linux kernel (version 3.8):

```
1  struct list_head {
2          struct list_head *next;
3          struct list_head *prev;
4  };
5
6  static void list_del(struct list_head *entry)
7  {
8            entry->next->prev = entry->prev;
9            entry->prev->next = entry->next;
10 }
```

This function essentially removes the given entry from its list. If the attacker controls the `next` and the `prev`

field from the entry to be deleted, she essentially can trigger an arbitrary 8-byte write on a 64-bit architecture. In particular, she can write the value of `prev` into the memory address `[next + 8]` (Line 8) and the value of `next` into the memory address `[prev]` (Line 9). To use this code fragment for a dynamic hook, the attacker could, for instance, modify a specific entry within the system and set its `prev` pointer to point to the return address of the `list_del` function and its `next` pointer to point to attacker-controlled code. When the entry is deleted, the `list_del` function will then, while processing the malicious pointers, overwrite its own return address and activate the code of the attacker on its return.

The example code above was selected as the `list_del` function is used throughout the Linux kernel and demonstrates the arguments stated above. In general, this function is not exploitable by an external attacker, as the entries that are used by the function are created by other internal functions within the kernel. While these functions initialize the values of the pointers correctly, an attacker that controls the kernel can modify them arbitrarily, thus creating a new vulnerability. The `list_del` function does not expect the manipulated values and uses them without checks. This enables an attacker to conduct an arbitrary 8-byte write, which is not enough to introduce shellcode into the system, but is sufficient to transfer the control flow to a previously prepared code region. In addition, the attacker is not hindered by any of the protection mechanisms used by the Linux kernel, since she can disable $W \oplus X$ for her code[1], does not need to overwrite the stack canary, and knows the address of her code or can calculate the address of the location of the return address[2]. Finally, since the `list_del` function is invoked by many other functions within the kernel, a dynamic hook within this function is very effective.

3.3 Suitable Vulnerabilities

In principle, any kind of vulnerability can be used to implement a dynamic hook. In this paper, however, we will, for the sake of simplicity, focus on *n-byte* writes, sometimes also referred to as *write-what-where* primitives, such as the one presented in the previous example. Such n-byte writes enable an attacker to modify *n* bytes at an arbitrary memory location. In our example, the attacker controls an *8*-byte write to an arbitrary memory address. In x86-assembly, n-byte writes are essentially a memory `mov` instruction for which the source and the destination operand can be controlled by an attacker. An example of a potential 8-byte write vulnerability in Intel assembly syntax is the following instruction:

```
mov [rax], rbx
```

If the attacker can control the contents of `rax` and `rbx` at the time the instruction is executed, she can misuse it for a dynamic hook. It goes without saying that such instructions appear frequently within software. In the Linux 3.8 kernel binary, for instance, we found more than 103,000 `mov` instructions similar to the one shown above that can potentially be abused for an 8-byte write. This corresponds to about 5% of all instructions (1,976,441) within the tested Linux kernel binary (Linux 64-bit 3.8 kernel). Note that this does not include the approximately 58,000 one, two, or four byte write instructions. Together, this equates to a total of 8% of all instructions that can potentially be used to realize a dynamic hook.

3.4 Types of Dynamic Hooks

Generally speaking, there are two different types of dynamic hooks: *dynamic control hooks* and *dynamic data hooks*. The former target the control flow of the victim application and can be used as an alternative to traditional hooks since they enable an attacker to intercept events within the application. Dynamic data hooks, on the other side, do not target control data, but rather other critical data structures within an application. As an example, consider that an attacker wants to install a backdoor. For this purpose, she places a dynamic hook into a control path that can be triggered from userland such as a specific system call. However, instead of changing control data, this dynamic hook will upon invocation directly overwrite the credentials of a predefined process and elevate its privileges to *root*. Since the task credentials are usually a data value, this can be achieved with a single memory write. Thus, instead of overwriting a return address, the attacker simply sets her hook to overwrite the memory location where the task credentials reside. As pointed out by Chen et al. [10], such non-control data attacks can be quite powerful.

While dynamic data hooks do not modify the control flow directly, they can be used to influence the control flow at a later point in time. Consider for instance data that resides in memory and is processed by a just-in-time compiler. If an attacker manages to overwrite this data with dynamic hooks before it gets compiled, she can influence the instructions that are introduced into the system, which can lead to arbitrary code execution [7].

[1] Note that this is necessary since the first write (Line 8) of the example will write the return address (`prev`) into the code (`[next + 8]`) of the attacker. However, this is not a problem in practice, as the attacker can set her code to be writable and executable. In fact, this is even the default for memory allocated in the Linux kernel via `kmalloc`.

[2] The location of the return address depends solely on the address of the kernel stack and the size of the current function's stack frame. Both values are known to the attacker as we will describe in Section 4.2.

3.5 Properties of Dynamic Hooks

Components. Dynamic hooks always consist of two integral components. On the one hand, there is the instruction that activates the hook, which we refer to as the *trigger*. In the case of an 8-byte write, the trigger is the `mov` instruction that conducts the write on behalf of the attacker. Every path that leads to the execution of the trigger is referred to as a *trigger path*. On the other hand, there is the data that was manipulated by the attacker and encodes the malicious action that the attacker wants to conduct. This is the *payload* of the hook. For n-byte writes, the payload usually consists of two manipulated pointers: the *destination pointer*, which contains the address that will be written to and the *source pointer*, which specifies the value that will be written.

Binding. While the same trigger can be shared among different dynamic hooks, each hook in general requires its own payload. The reason for this is that the payload contains the actual data that specifies the control transfer. This data, however, will only be valid in a particular *context*. To overwrite a specific return address, for example, we must first be able to predict its exact location. This requires us to know the exact path leading to the use of the payload by the trigger. In practice, this means that a payload and thus the dynamic hook is usually closely *bound* to a specific execution path. The closer the connection between an execution path and a dynamic hook, the better the control of the attacker over the hook.

In an ideal situation, a dynamic hook is *exclusively* bound to a specific execution path. In this case, the payload of the hook is *only* processed in the execution path that leads to its trigger. This enables the attacker to predict possible modifications applied to the payload before its use in addition to the state of the machine at the time of the exploitation with high probability, since she must only consider a single execution path. Similar to traditional exploits, this is essential information that is required to be able to setup a dynamic hook correctly. After all, the attacker needs to correctly predict the exact address of the control data, which should be overwritten and overwrite it with the precise address of the target code region. Without knowing the exact layout of the stack as well as the transformations that may be applied to the payload before its use, this is a hard task.

If there are multiple paths that use the payload, the dynamic hook is only *loosely* bound to the path leading to the trigger instruction. The more execution paths the payload affects, the more difficult it will become for an attacker to control the hook. On the one hand, this is due to the fact that it will become increasingly difficult to predict the necessary memory addresses and transformations as has been described above. On the other hand, the more functions access the actual payload that the attacker modified, the more likely it will be that the hook introduces side effects into the application that may lead to unexpected behavior and crash the application. Consider, for instance, that an entry that is used by the `list_del` function has been modified to act as payload for a dynamic hook. If the same entry is used by a different function to iterate through all elements within the list, this will most likely lead to a crash of the system as the `prev` and the `next` pointer do not point to the previous and next element, respectively, as would have been expected.

Coverage. Another important property of a dynamic hook is coverage: as dynamic hooks should be closely bound to the execution path containing the trigger, it is essential that this triggering path is *always* executed when the target event that should be hooked is invoked. In this case, the dynamic hook provides *full* coverage. Otherwise, the hook may only be able to intercept *some* execution paths of the target event, but not all. In that case, the hook has only *partial* coverage and must thus be combined with other dynamic hooks to be able to achieve *full* coverage of the target function. Note that while binding is a property of the payload of the hook, coverage is a property of the trigger instruction.

3.6 Automated Path Extraction

So far we have discussed the concept of dynamic hooks and provided an overview of the different types of dynamic hooks and their properties. However, the creation of a dynamic hook still remains a manual process, which can—as in the case of traditional exploitation—be very time-consuming and error-prone especially for complex binaries such as modern OS kernels. We now describe how paths for dynamic hooks can be obtained automatically for a given binary. This is essentially a two-step process: In the first step, we make use of static program slicing [40, 45] to extract potential paths that could be used for a dynamic hook. In the second step, we then employ symbolic execution [17, 34] to verify the satisfiability of the paths and to generate detailed information for their exploitation.

3.6.1 Program Slicing

To find possible locations for dynamic hooks within an application, an attacker has to find triggers that make use of a payload that she can control. Since trigger instructions can be as simple as a memory move, there usually exist many triggering instructions in many paths of the application. To identify whether a particular trigger instruction can be used for a dynamic hook, it is necessary

to analyze the data flow that leads to the particular instruction. One technique that can be used for this purpose is static program slicing [40, 45].

The basic idea behind static program slicing is to traverse back through the control flow graph (CFG) of an application starting from a *sink* node and to extract each node that directly or indirectly influences the values used at the sink. Applied to the problem of finding dynamic hooks, static program slicing thus allows us to determine where the values of the *source* and the *destination* pointer in an n-byte write originate. To achieve this, we first identify all potentially vulnerable `mov` instructions within a given binary. These are essentially all `mov` instructions which move a value contained within a register to a memory location specified by another register. In the next step, we then traverse the CFG of the binary backwards at the assembler level until we encounter the first instruction that modifies the source register of the move. We record this instruction and continue with our backward traversal. Instead of looking for instructions that modify the source register of the original move, however, we will from here on search for instructions that modify the source register of the last instruction we recorded. If we continue this process, we eventually obtain the register or memory location where the value that is later on contained within the source register originates. We then repeat the process for the destination register. All the instructions that we recorded using this method form a *slice* of the binary. Each slice contains all the instructions that affect a given vulnerable `mov` instruction.

We implemented a *slicer* which is capable of extracting potential paths that could be used for n-byte writes from a 64-bit Linux or Windows kernel binary. The implementation of the slicer is based on the disassembler *IDA Pro* [14]. In particular, we make use of the CFG that IDA provides to perform the above described *static interprocedural def-use analysis*. Starting from each trigger, we perform a breadth-first search in a backwards direction. We hereby make use of a register set to conduct the actual analysis. Initially, this register set consists of the source and destination register. Whenever we encounter an instruction that modifies a register included within the register set, we add the source register of the instruction to the set and remove the modified register. Since we walk backwards through the instruction stream, this effectively allows us to record and track the *def-use chains* for the source and destination register. In addition, we record all instructions that we visit along the way, in order to be able to reconstruct the path that we explored in case we consider it to be potentially exploitable.

The challenge that remains to be solved is to determine whether a slice can be used for a dynamic hook or not. To address this problem, we must know whether the registers in the vulnerable move can be controlled by an attacker. We consider this to be the case if the values of the source and destination register originate from a *global* variable. The reasoning behind this approach is that the data used within the move in this case stems from a persistent location. Consequently, to control the final `mov` instruction, an attacker can modify the pointer chain starting from the global variable.

To identify global variables in the kernel, we assume that each access to a fixed address or the *Global Segment register (GS)* constitutes an access to a global variable. The reason for the latter is that both the Linux and Windows kernels store important global variables that are valid for a particular CPU within a memory region pointed to by this register. For instance, both Linux and Windows store the address of the `task_struct` (`gs:0xc740`) or the `_ETHREAD` (`gs:188`) structure of the process that is currently executing in this memory region.

If both the source *and* the destination register originate from a fixed address or the memory region pointed to by GS, we consider the path to be potentially exploitable and record it such that it can later on be used as input for the symbolic execution engine.

3.6.2 Symbolic Execution

Symbolic Execution is a well-known program analysis method that has been proposed over three decades ago [8, 17]. The basic idea of symbolic execution is to treat input data of interest as symbols rather than concrete values. These symbols can represent any possible value and as we walk over the code of a program, the values become constrained. Branches, for instance, set up conditions that constrain symbolic variables. Each of these conditions can be represented as a logical formula which can then be fed into an SMT solver to obtain concrete values that satisfy the path conditions. A profound introduction is available in the literature [25, 34].

We use forward symbolic execution to verify the satisfiability of our sliced paths and to produce detailed information for the creation of the dynamic hooks. In the process, we utilize the VEX IR, which is a RISC like intermediate representation with single static assignment (SSA) properties, deeply connected to the popular Valgrind toolkit [24]. Due to space limitations, we refrain from discussing this intermediate language in detail.

To verify satisfiability, we transform each basic block of the sliced path into VEX IR code and execute the code symbolically. The translation to VEX IR is achieved by utilizing a python framework called *pyvex* [36]. We dismantle every VEX statement that we obtain from *pyvex* and link the components of the statements into our own data structures. These data structures are used to walk over the VEX code and by doing so, we semantically map the statements to Z3 expressions. Z3 is a theorem

prover developed at Microsoft Research that we use to solve our formulas [23].

As we walk over the VEX code of our sliced paths, we also keep track of three global contexts, i.e., a memory context, CPU context, and the current jump condition. Each context consists of Z3 expressions that semantically mirror the current state of the execution. Additionally, each basic block also keeps track of temporary VEX IR variables in SSA form. By constant propagation, we use these variables to resolve source and destination. Each store, load, and register set statement updates the corresponding context in form of Z3 expressions. Once we hit a jump condition, we ask the solver whether we can take the jump according to our context. If no solution exists, we can filter out the path. An unsatisfiable set of formulas stops execution of the current path, and we move on with the next slice.

At this point it is worth mentioning that we do not use symbolic execution in the traditional sense to achieve code coverage. Our main goal is to check whether we can walk down our paths and to determine what value sets lead us to the end of the slice. We use the symbolic formulas to generate detailed information about the controlled registers at the time the vulnerability is triggered as well as the jump conditions that must be fulfilled to actually reach the trigger. By processing over the VEX code, the solver also gives us possible values to set.

4 Experiments

Based on the slicer and the symbolic execution engine, we created a prototype that we used to automatically extract paths for dynamic hooks in a fully patched Windows 7 SP1 64-bit kernel and a Linux 64-bit 3.8 kernel. We chose this approach for three main reasons. First and foremost, since malware nowadays generally attacks the kernel, this approach allowed us to test the prototype in a realistic scenario. Second, kernel binaries are especially complex, which makes them well suited for a thorough test of our implementation. Finally, by targeting Windows and Linux, the experiments show that the proposed mechanism is applicable to two of the most popular OSs.

In the following, we first discuss the results that we obtained by providing detailed statistics about the automatically extracted paths for both kernels. To demonstrate how useful the prototype is when it comes to the actual creation of the hooks, we also describe three concrete POCs for dynamic hooks that we created based on the information that the prototype provided.

4.1 Automated Path Extraction

As stated above, we tested our prototype with a fully patched Windows 7 SP1 64-bit kernel and a Linux 64-bit 3.8 kernel. The goal of the experiment was to automatically extract trigger paths that could then either be used by a human expert to manually design dynamic hooks or to automatically generate exploits. Table 1 provides an overview of the obtained results.

At first, we determined the number of instructions contained within both kernel binaries for reference. In the next step, we obtained the number of potentially exploitable 8-byte `mov` instructions. In the process, we only counted those `mov` instructions that move data from one general purpose register into a memory location specified by another general purpose register with the condition that the involved registers were neither `rbp` nor `rsp`. The reason for this restriction is that our prototype implementation currently does not support a memory model, meaning that we cannot track memory store and load operations in our slicer, which is why we currently ignore any path that requires this functionality. We will cover this limitation in more detail in Section 5.3. As Table 1 shows, about 2 % of all instructions within the tested kernels are `mov` instructions that fulfill this criteria.

Next, we used the slicer to extract potentially exploitable slices for each of the identified moves. In case of Linux, the slicer considered about 4% of the `mov` instructions as potentially exploitable, while on the Windows side about 20% of the `mov` instructions were marked as possibly exploitable. We assume that the significant difference between Windows and Linux stems from the fact that Linux has substantially more `mov` instructions that store or load data from memory (61,651 vs 37,272). Since the slicer does not support a memory model, it will abort whenever such a `mov` instruction is part of a def-use chain. Due to their number, this scenario is more likely to occur on Linux than on Windows.

Finally, we symbolically executed each of the obtained slices. In total, this led to 566 exploitable paths for Linux and 379 exploitable paths for Windows. The symbolic execution engine thereby produced the required value for each conditional jump within the path and detailed information of the vulnerable `mov` instruction. In particular, the output[3] specifies exactly which memory addresses must be modified in what way to pass the conditional jumps and where the source and destination values are located, respectively. This information can directly be applied to generate exploits or to manually create a dynamic hook as we will show in the next section.

4.2 Prototypes

We now present three concrete examples of dynamic hooks to illustrate the capabilities and properties which have been discussed throughout the paper. We created

[3] An example of the output is shown in Section 4.2.3.

OS	*Size*	*Instructions*	*8-byte moves*	*Slices*	*Paths*
Linux 3.8 64-bit (`vmlinux`)	18.8 MB	1,976,441	42,130 (2.1%)	1753 (4%)	**566 (32%)**
Windows 7 SP1 64-bit (`ntoskrnl.exe`)	5.3 MB	1,330,791	26,694 (2.0%)	5450 (20%)	**379 (07%)**

Table 1: Overview of the 8-byte moves, the potentially exploitable slices, and the exploitable paths according to the symbolic execution engine for the analyzed Linux and Windows kernels.

these examples based on the output provided by our prototype. The first and the third example focus on a dynamic control hook, while the second example demonstrates a dynamic data hook. To ease the understanding of the examples, all hooks leverage a trigger instruction within the `list_del` function (as explained in Section 3.2) or its Windows equivalent. The first two hooks were implemented for Linux 3.8 and an Intel Core i7-2600 3.4 GHz CPU. To demonstrate that the proposed concept is similarly applicable to Windows, the third hook was implemented on a fully patched version of Windows 7 SP1 running on the same CPU.

4.2.1 Dynamic Control Hook: Intercepting Syscalls

A common functionality that kernel level malware requires is the possibility to intercept system calls. In this example, we show how a single dynamic hook can be used to intercept *all* system calls for a particular process. To achieve this, the hook is placed into the execution flow of the *system call handler*, which is—independent of the system call mechanism that is used (i.e., interrupt-based, sysenter-based, or syscall-based)—invoked whenever a system call on the x86 architecture is executed. The main purpose of the syscall handler is to invoke the actual system call by using the system call number as an index into the system call table.

Similar to other functions within the kernel, the system call handler can be audited for debugging reasons. Auditing can be enabled or disabled within the flags field of the `thread_info` struct associated with each process. By setting the `TIF_SYSCALL_AUDIT` flag, every system call conducted by a process will also lead to the invocation of the auditing functions. In particular, the function `__audit_syscall_entry` will be executed before the invocation of a system call and the function `__audit_syscall_exit` will be executed after the system call, but before the system call handler hands control back to user space. In our POC, the dynamic hook is set within the `__audit_syscall_exit` function.

When syscall auditing is enabled, the `__audit_syscall_entry` function records information about the system call such as the syscall number and the arguments of the syscall within the audit context of the process. The purpose of the `__audit_syscall_exit` function is to reset the audit context of the task before the system call returns. In the process of resetting the audit context, this function invokes the inline function `audit_free_names`, which resets the `names_list` within the audit context:

```
1 static inline void audit_free_names(
2         struct audit_context *context) {
3     ...
4     list_for_each_entry_safe(n, next,
5           &context->names_list, list) {
6             list_del(&n->list);
7     }
8     ...
9 }
```

The `audit_free_names` function essentially iterates over the `names_list` of the audit context (Line 4) and deletes every entry within the list (Line 6). Consequently, if we control the `names_list`, we can control the entry that is passed to the `list_del` function, which in turn allows us to exploit its vulnerability. As the `names_list` is not modified by the `__audit_syscall_entry` function or anywhere else in the kernel[4], the attacker is free to modify it in any way she wants. That is, the `names_list` structure is exclusively bound to the execution path within the syscall handler that we use for our dynamic hook.

While the `names_list` structure seems to be perfectly suited for a dynamic hook, the triggering path places additional constraints on the hook. The problem arises due to the fact that the `list_del` function is contained within a loop that iterates over all entries within the `names_list` list (Line 4). To iterate through the list, the loop will essentially follow the `next` pointer in every entry until one of them points back to the first element in the list, which is `&context->names_list`. Since we want to modify the `next` and the `prev` pointer of an entry within the list to conduct an arbitrary 8-byte write, we have to take this problem into account and assure that the list iteration will eventually terminate. To achieve this we initialize the audit context as shown in Figure 1.

The basic idea behind this setup is to make use of a special address, referred to as a "magic address", that is a valid memory address, but at the same time contains valid x86 instructions. Due to little-endian byte order, these valid instructions must be contained in re-

[4]While there are other functions in the kernel that try to access the `names_list`, these attempts can be blocked by setting the first member within the audit context (`dummy`) to one.

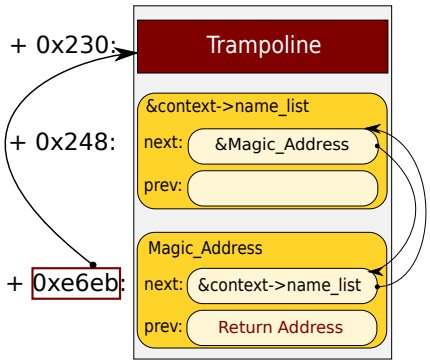


Figure 1: The audit context structure that the attacker uses to set a dynamic hook within `audit_free_names`.

verse order within the address. In Figure 1, the instruction encoded into the address is a negative relative jump (0xe6eb (address) $\Rightarrow$ 0xebe6 (instruction)) that will upon execution transfer control to a trampoline, that then transfers control to an arbitrary address. Initially when the loop begins iterating over the `names_list`, it follows the `next` pointer to the first entry within the list, which is located at the magic address. The `next` pointer stored at the magic address will in turn point back to the `names_list`, thus fulfilling the loop condition. However, before the loop exits, the first entry in the list (located at the magic address) is processed by the `list_del` function. Since the `prev` pointer of this entry points to the location of a return address, the `list_del` function will overwrite this return address with the value stored in the `next` pointer ($prev \rightarrow next = next$), which points to `&context->names_list`. Consequently, as soon as the return address is used, control will be transferred to the address of `&context->names_list` where the magic address is stored, leading to the execution of the magic address and the activation of the trampoline code. Note that the hook requires the audit context region created by the attacker to be writable and executable, since the `list_del` function conducts *two* write operations as has been described in Section 3.2. This is not a big problem in practice, since every memory region allocated with `kmalloc` is by default writable and executable.

The final problem that remains is which return address we are actually going to overwrite and how we can predict its location. As previously stated, the syscall handler is invoked before every system call and it will invoke the actual function handling the syscall. Thus, if we know the stack frame size of the syscall handler and the location of the kernel stack, we can predict where the return address of the function that is invoked by the syscall handler resides. The stack frame size can be obtained from the assembler code of the syscall handler, while the location of kernel stack can be obtained from a kernel variable (`get_cpu_var(kernel_stack)`). The target return address will then reside at `stack_frame_size + get_cpu_var(kernel_stack)`.

Summary. A dynamic control hook for intercepting all system calls for a particular process can be placed in the `audit_free_names` function. To ensure that execution passes through this function, we set the `TIF_SYSCALL_AUDIT` flag within the `thread_info` struct of the target process. In the next step, we modify the audit context of the target process in the way described above and use a trampoline to control the execution flow. This enables us to reliably divert the control flow at run-time. The resulting dynamic hook will have full coverage and be exclusively bound to the execution path leading to the `audit_free_names` function.

4.2.2 Dynamic Data Hook: Installing a Backdoor

In the second example, we demonstrate the possibilities of dynamic data hooks. In particular, we show how a dynamic data hook can be used to install a backdoor within a Linux system that is capable of elevating the task rights of a predefined process to *root*. For this purpose, we leverage the `ptrace` system call, which enables one process to attach to another process for debugging reasons. To install the backdoor, we simulate that a process used the `ptrace_attach` system call to attach to the target process (i.e. the process that will contain the hook). This is achieved by manually applying the changes that the `ptrace_attach` function conducts to the internal data structures of the target process. Most importantly, the `state` field of the task must be updated to include `__TASK_TRACED`, the `ptrace` field within the task must be set to 1, and the `parent` field must be set to the process which will later trigger the backdoor. We will defer the discussion of this last change for the moment and explain it in more detail later on.

Once the changes of the `ptrace_attach` function have been simulated, it is possible to invoke the `ptrace_detach` function on the so prepared process. The execution of this function eventually leads to the invocation of the `__ptrace_unlink` function, which in turn invokes the `list_del` function using the `ptrace_entry` pointer within the target process as argument:

```
1 void __ptrace_unlink(
2       struct task_struct *child) {
3       ...
4       list_del(&child->ptrace_entry);
5       ...
6 }
```

To use this code fragment for a dynamic data hook, we modify the $ptrace\_entry \rightarrow next$ pointer and the

ptrace_entry → *prev* pointer of the target process. This enables us to conduct an arbitrary 8-byte write when the `list_del` function is invoked during the execution of `ptrace_detach`. In particular, we set the `prev` pointer to point to the task credentials that we want to override and the `next` pointer to an address that is writable and ends with four zero bytes. To understand this, we have to take a look at the Linux task credential structure, which defines the access rights of a process:

```
1  struct cred {
2      ...
3      kuid_t  uid;  /* real UID */
4      kgid_t  gid;  /* real GID */
5      kuid_t  suid; /* saved UID */
6      kgid_t  sgid; /* saved GID */
7      kuid_t  euid; /* effective UID */
8      kgid_t  egid; /* effective GID */
9      ...
10 };
```

Each task contains three pairs of access rights and each access right pair consists of a user id and a group id. Most important for us is the effective user id (`euid`), which specifies the effective access rights of a process. Since the root user in Linux generally has the user id zero, our goal is to overwrite the `euid` field, which has a size of 4 bytes, with zeroes. If we choose an address for the `next` pointer that has its lower 32-bits set to zero and additionally set the `prev` pointer to point to the `euid` field of the process whose privileges we want to elevate, we will—due to the little endian byte order—overwrite the `euid` (*prev* → *next* = *next*) field with zeroes and thus set the access rights of the process to root. However, because the `list_del` function will also write the `prev` pointer into the address of `[next + 8]` (*next* → *prev* = *prev*), we have to ensure that the address used within the `next` pointer points to a writable memory region that does not contain crucial data. A possible address that can be used for this purpose is `0xffff880000000000` since this address usually points to the first 8-bytes of the physical memory of the machine, which is not used by the Linux kernel. Finally, note that we will also override the `egid` of the process with the upper 32-bits of the address in the next pointer. This will, however, not affect the process as long as it has a valid `euid`.

We can now set up a dynamic hook as follows: First, we need to select a target process that remains running on the system as it will contain the above described dynamic hook. Good candidates are therefore background daemons such as the SSH daemon. Second, we need to specify the victim process whose privileges we want to elevate and setup the dynamic hook within the target process. Since we need to know the address of the task struct of the victim process in order to be able to set the `prev` pointer to its `euid` field, this process also needs to remain running. A good choice in this case could, for instance, be a shell process within a screen session.

To activate the backdoor, we need to call the `ptrace` syscall with the `PTRACE_DETACH` argument on the target process. However, the backdoor cannot be activated by any process because only the tracing process can detach from the traced process. Since we simulate the changes conducted by `ptrace_attach`, the process which can execute the `ptrace_detach` call, is the process that we specify as `parent` during the setup of the dynamic hook. While this ensures that the backdoor cannot be triggered by accident, this requires us to specify the process that triggers the backdoor when we setup the dynamic hook. The easiest way to solve this problem is to specify the victim process as parent of the target process. In this case the victim, whose privileges will be elevated, can trigger the backdoor itself.

Summary. A dynamic data hook can be used to implement a backdoor that can be triggered from user space with arbitrary access rights. In our example, the backdoor is closely bound to the process that was specified as the tracing process and to the execution path within `ptrace_detach`. In addition, the hook only provides partial coverage as only the detach call to a specific process will trigger it, which is desired behavior in the case of a backdoor.

4.2.3 Dynamic Control Hook: Process Termination

To show that the proposed hooking concept can be applied to other OSs as well, we will in our final example present a dynamic control hook that we implemented on a fully patched version of Windows 7. In particular, the hook is capable of intercepting the termination of an arbitrary process, which can, for instance, be useful in situations where a malicious process on the system is found and terminated by a security application or the user. Due to the hook, the malware would be notified of this event and could react to it.

When a process is exiting on Windows 7, the function `NtTerminateProcess` is invoked which in turn invokes various cleanup functions that prepare the termination of the process. One of these functions is `ExCleanTimerResolutionRequest`. To support a wide range of applications, Windows provides processes with the possibility to request a change to the system's clock interval [32]. This enables programs that have a demand for a faster response time to decrease the clock interval and thus to increase the number of clock-based interrupts. When a process emits such a request, the process is added to the `TimerResolutionLink` list, which is used by the OS to manage all timer resolution changes. As the name suggests, the purpose of the `ExCleanTimerResolutionRequest` function is to remove processes from the man-

agement list once they exit. Our automated path extraction tool discovered the following path within this function:

```
1  ———SLICE———
2  0x000000014042c396   mov      rax, gs:188h
3  0x000000014042c39f   mov      rbx, [rax+70h]
4  0x000000014042c3c6   mov      rcx, [rbx+4A8h]
5  0x000000014042c3cd   mov      rax, [rbx+4B0h]
6  0x000000014042c3d4   mov      [rax], rcx
7  0x000000014042c3d7   mov      [rcx+8], rax
8
9  ———SYMBOLIC———
10 Jump Condition in: BB_0x14042c390
11 Concat(0x0, Extract(0x1f, 0x0,
12 MEM[RBX+0x440])) >> Concat(0x0, 0xc) &1 == 0
13
14 CPU CONTEXT/CONTROLLED REGISTERS
15 RCX -> MEM[MEM[MEM[0x188+GS]+0x70]+0x4a8]
16 RAX -> MEM[MEM[MEM[0x188+GS]+0x70]+0x4b0]
```

To remove a process from the TimerResolutionLink list, the ExCleanTimerResolutionRequest function obtains the forward and the backward pointer (Line 4 and Line 5) from the EPROCESS structure of the process and performs the discussed list delete operation (Line 6 and Line 7). The only prerequisite for this path is that the 13th least significant bit of the memory word at location EPROCESS+0x440 is not set (Line 11). By manipulating this memory word and the pointers, which are located within in the EPROCESS struct of the process at offset 0x4A8 (Line 4) and offset 0x4B0 (line 5) respectively, we can thus perform an arbitrary 8-byte write and change the control flow. In our POC we set the forward pointer (rcx) to point to our shellcode and the backward pointer (rax) to point to the return address of ExCleanTimerResolutionRequest. Just as in the case of our first example, the location of the latter can be obtained by subtracting the sum of the stack frames of the invoking functions from the start address of the kernel stack, which is stored within the InitalStack variable contained within the KTHREAD structure of the thread of the process. Similarly, the area where the shellcode resides must be writable *and* executable. On Windows, we can allocate such a memory region by invoking the ExAllocatePoolWithTag function with the argument NonPagedPoolExecute.

One last problem that remains, however, is that the TimerResolutionLink entry structure of a process is unfortunately not exclusively bound to the path of our dynamic hook, since the TimerResolutionLink list is also used by other functions such as ExpUpdateTimerResolution. The solution to this problem is quite simple, though: since the TimerResolutionLink list is not critical for the execution of a process and the ExCleanTimerResolutionRequest function does on top of that not iterate through the list, but rather accesses the forward and backward pointers directly, we can simply remove the entry from the linked list. As a result, the manipulated entry will no longer be processed by other management functions, which will bind the TimerResolutionLink entry structure exclusively to our trigger path. In our experiments, removing processes from the TimerResolutionLink list did not affect their execution in any way. The proposed dynamic hook therefore serves as an example that an exclusive binding of a hook payload must not be given by the target application, but can also be manually enforced by the creator of the hook.

Summary. By manipulating the TimerResolutionLink entry structure of a process in the way described above we can install a dynamic hook and intercept the termination of an arbitrary process on Windows. While the manipulated structure is by default not exclusively bound to the trigger path, the creator of the hook can enforce an exclusive binding manually by removing the manipulated entry from its linked list. In addition, the presented dynamic hook had full coverage in our experiments. It was even triggered if we forcefully terminated the process using the task manager.

5 Discussion

Up to this point, we have not discussed what kinds of transient control data exist. This is why it may seem to the reader that dynamic control hooks could be mitigated by protecting return addresses alone. In this section, we cover this topic in more detail and show that this is not the case. In addition, we cover possible countermeasures against dynamic hooks and review the limitations of the proposed hooking concept and our current prototype.

5.1 Transient Control Data

Instead of targeting *persistent* control data such as function pointers in the system call table, dynamic control hooks change *transient* control data at run-time. While return addresses are a popular example of *transient* control data, it is not the only kind of transient control data that exists. For instance, if a function allocates a local function pointer, this pointer will reside on the stack and not in the data segment or the heap. Instead of overwriting the return address, an attacker can in such a case similarly target the function pointer. While this is a rather unlikely scenario, it demonstrates a very important class of attacks where a local variable on the stack is changed to achieve the desired control flow change. This class of attacks is not restricted to function pointers alone. Consider, for example, the following code from the read sys-

tem call in the Linux kernel[5]:

```
1  struct fd {
2      struct file *file;
3      int need_put;
4  };
5
6  SYSCALL_DEFINE3(read, unsigned int, fd, char
7                  __user *, buf, size_t, count) {
8      struct fd f = fdget(fd);
9      ...
10     ret = f.file->f_op->read(f.file, buf,
11                              count, pos);
12     ...
13 }
```

In this case, a local structure (`struct fd f`) is allocated on the stack (Line 8). The structure contains a pointer to another structure (`struct file *file`), which in-turn contains a function pointer that is called in Line 10. With the help of a dynamic hook, an attacker could modify the pointer within the local structure (Line 2) and point it to an attacker-controlled structure instead. If she manages this before the function call in Line 10 is executed, this will effectively allow her to control the function call and thus enable her to arbitrarily change the control flow.

Instead of targeting a return address or a function pointer directly, the attacker in this scenario modifies a local pointer on the stack. This approach enables her to control any data that the local function accesses using this pointer. In the kernel, where objects in general are accessed through pointer chains, this represents a powerful attack vector, which effectively provides control over *any* object that the pointer references. Since similar code exists in many other functions within the kernel, this attack vector must be taken into account when one considers countermeasures against dynamic hooks.

5.2 Countermeasures

Dynamic hooks are installed by an attacker that already controls the application, which renders many of the existing defense mechanisms against exploits ineffective. However, while dynamic hooks are a powerful attack vector, there are, of course, countermeasures that can be used to reduce the attack surface. In the following, we first discuss possible countermeasures against dynamic control hooks, before we present defense mechanisms for dynamic data hooks.

Dynamic control hooks. What makes dynamic control hooks difficult to detect is that they do not permanently modify control data. Instead, their payload is hidden within non-control data and the actual control flow modification only occurs at run-time. This enables them to evade popular hook detection mechanisms such as HookSafe [43] or SBCFI [27], which only protect *persistent* control data, but ignore *transient* control data on the stack. However, at some point during the execution, dynamic control hooks must override control data in order to divert the control flow. Thus while a dynamic hook may be hidden at first, it will become visible when it is triggered. The resulting control flow change can potentially be detected using control flow integrity (CFI) and related approaches [1, 15, 20, 39, 42, 46, 48].

In order to detect dynamic control hooks with CFI, it is crucial that *every* control transfer of an application is verified. If a single control transfer is missed, this can potentially be abused by an attacker to install a dynamic hook. However, finding all possible control transfer instructions within complex software such as an OS kernel is a difficult problem. This is especially true if we consider attacks on transient control data such as the one present in the last section. Even worse, control transfer instructions can often have more than a single target. Consequently, one must not only identify all control transfer locations, but also all the possible targets of these transfers to avoid false positives. Additionally, if there are multiple possible targets for a given control transfer instruction, an attacker can still launch return-to-libc like attacks [37], which is a general problem of CFI mechanisms [48]. Finally, every check of a control transfer comes at a cost [39]: the more instructions we verify, the higher the overhead will be and for applications that are optimized for performance such as an OS kernel, even a small overhead can have a huge impact.

While current CFI approaches are not yet able to solve all of these problems, they certainly reduce the attack surface and make it more difficult to install dynamic control hooks. The results presented in this paper can help to further improve these mechanisms by using the discussed techniques to automatically extract possible triggers from a given application and adding additional checks to verify them. To increase the performance, one could make use of lazy control flow verification as proposed by Bletsch et al. [7]. This could result in an effective and efficient detection system, which might not be able to eliminate dynamic control hooks entirely, but will significantly raise the bar for an attacker.

Dynamic data hooks. The defense mechanisms discussed above make use of the fact that dynamic control hooks have to eventually modify the control flow. That is, the detection mechanisms do not focus on the hook itself, but rather target the *effects* of the hook's invocation. The idea behind dynamic data hooks is to complicate detection even further by modifying non-control data instead of control data. As a result, defenders can no

[5]For better readability we directly included the `vfs_read` function into the `read` system call. In the actual code the function call in Line 10 will occur in the `vfs_read` function.

longer concentrate on the control flow of the application alone, but rather have to detect integrity violations within the data of the application.

Verifying the integrity of data structures is a difficult problem. Petroni et al. [26] were the first to propose a general architecture for the detection of kernel data integrity violations. Since then various systems have been proposed that try to detect or prevent malicious modification of kernel data structures [9, 15, 19, 30, 33]. What is common to all these approaches, however, is that they only *enforce* integrity checks, but leave the *creation* of the actual integrity constraints to a human expert. To the best of our knowledge, the only approach that tries to generate integrity constraints for kernel data structures automatically is Gibraltar [4]. While this approach provides a good starting point and could support a human expert in the creation of integrity constraints, the authors acknowledge that the generated invariants are "neither sound nor complete" [4]. Creating reliable and evasion-resistant integrity constraints is, however, the basis for the detection of dynamic data hooks.

To be able to effectively protect kernel data structures, additional research in the field of automatic integrity constraint generation is required. Techniques that are able to generate signatures for kernel data structures such as the ones presented by Lin et al. [21] or Dolan-Gavitt et al. [12], could thereby provide a good starting for further research, as the generated signatures could potentially be used to infer integrity invariants. In the meanwhile, initial defense mechanism could use systems such as HookMap [43] or K-Tracer [18] in combination with the techniques presented in this paper to generate integrity constraints for known dynamic hooks that can then be enforced by one of the systems mentioned above.

Summary. While the threat of dynamic control hooks can potentially be reduced with the help of (kernel-level) CFI mechanisms, dynamic data hooks pose a difficult problem that cannot be easily solved. To detect dynamic data hooks, reliable integrity constraints are required that allow the automatic verification of the kernel data regions. Until these constraints are available, one could reduce the attack surface with the help of manual integrity specifications or by automatically creating integrity constraints for known attacks.

5.3 Limitations

Dynamic Hooks. Dynamic hooks essentially face two limitations. First and foremost, not every function may contain a vulnerability that can be used to implement a dynamic hook. In contrast, it is likely that there are functions which are immune against the attack. However, this is not a big problem in practice: if a particular function cannot be hooked directly, it may still be possible to intercept calls to the function by hooking a function that immediately precedes or follows the function in the execution flow. After all, not every function contains a function pointer either. Function pointer hooks have nevertheless been proven to be very effective in practice.

Second, similar to traditional exploits, a dynamic hook may face restrictions that are caused by the vulnerability it is exploiting. For instance, specific hooks such as the one presented in our first prototype (see Section 4.2.1) may require that certain memory areas are writable and executable. Depending on its restrictions, a dynamic hook may therefore not be suitable for every scenario. This, however, heavily depends on the particular hook.

Automated Path Extraction. While our prototype already produces very valuable paths that can be used to implement powerful dynamic hooks as we have shown in Section 4.2, it also faces some limitations. First, our slicer does not yet support a detailed memory model. As a result, we are unable to find dynamic hooks on paths where registers, which are currently monitored, are loaded with values from the stack. This situation frequently occurs when subfunctions are called. In this case, the calling function often stores register values temporarily on the stack to guarantee that they are not overwritten by the subfunction. During our experiments, the slicer ignored 79,853 such paths due to this restriction.

Second, the symbolic execution engine currently only handles a subset of the available x86 instructions. Most importantly, it is unable to handle some instructions that are a ring-0 privilege. This is, however, a restriction in the VEX intermediate language. In the experiments we conducted, this led to 949 (55%, Linux) and 4,908 (90%, Windows) paths that could not be verified.

Finally, the slicer and the symbolic execution engine currently do not consider the properties of *binding* and *coverage*, while determining whether a path could be used for a dynamic hook or not. Consequently, not all of the paths extracted by our prototype will be suited for the implementation of a dynamic hook. As described in Section 3.5, especially the property of binding can be a limiting factor. If a payload is only loosely bound, it is likely that the hook will introduce side effects that can lead to a crash of the system. Determining automatically whether a path has exclusive binding or full coverage is difficult though. As the discussed POCs show, even payloads that initially seem unsuited for the implementation of a dynamic hook can through subtle manipulations of the involved data structures yield very reliable hooks. To designate the binding of a payload, we thus not only have to identify whether a payload is used in multiple locations, but we also have to establish how many of those usages can be controlled by the attacker. This requires

a profound semantic understanding of the data structures and functions involved.

6 Related Work

To the best of our knowledge, Petroni et al. [27] were the first to consider the hooking of transient control data. However, their work is primarily focused on the detection of *persistent* control flow modifications. Attacks on transient control data are thereby only mentioned as a limitation of their system. Hofmann et al. [15] presented a "return to schedule" rootkit that overwrites return addresses of sleeping processes to periodically invoke itself and evade hook detection mechanisms. While related to our work, this approach does not leverage exploitation techniques to change the control of an application at runtime. As a consequence, the technique only enables the rootkit to reschedule itself, but it does not allow it to intercept events within the system, which is the actual goal of a hooking mechanism.

In addition, there has also been a lot of work concerned with the possibilities of non-control data attacks. Chen et al. [10] were the first to demonstrate that non-control data attacks are indeed a dangerous and realistic threat. Sparks and Butler [38] presented DKOM as a general mechanism to hide objects within kernel space. Baliga et al. [5] extended this work and presented another class of stealthy attacks that do not have the goal of hiding objects, but rather target crucial kernel data structures to subvert the integrity of the system. Finally, Prakash et al. [28] discussed the manipulation of semantic values in the kernel to evade virtual machine introspection (VMI).

7 Conclusion

In this paper, we presented a novel hooking concept that we coined dynamic hooks. The main insight behind this concept is that existing hooking mechanisms are based on the permanent modification of *persistent* control data. As a consequence, the resulting hooks are constantly evident within the system and can be detected by verifying persistent control data alone.

Dynamic hooks solve this problem by targeting *transient* control data at *run-time*. This is achieved by applying exploitation techniques to the problem of hooking. To install a dynamic hook, an attacker will modify the internal data structures of an application in such a way that its usage will trigger a vulnerability at run-time. The hook thereby only consists of the modified data as well as the exploitation logic. This results in a powerful attack model with a wide range of possibilities as the attacker can make use of the entire arsenal of exploitation mechanisms to achieve her goal. At the same time, the hook will remain hidden in non-control data until it is triggered, which makes dynamic hooks not only powerful, but also difficult to detect in practice.

To show the applicability of the approach, we implemented a prototype that is capable of automatically extracting paths for dynamic hooks from recent Linux and Windows kernels. The experiments that we conducted prove that dynamic hooks are not only a dangerous, but are also a realistic threat that can be applied to practical scenarios such as system call hooking and backdooring. In future work, we plan to further improve our prototype implementation and to make use of it to generate integrity constraints instead of attack vectors that can then be used for the reliable detection of dynamic hooks.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive and valuable comments. This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant 16BY1207B (iTES).

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[2] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *Symposium on Network and Distributed System Security (NDSS)* (2011).

[3] BAILEY, M., COOKE, E., JAHANIAN, F., WATSON, D., AND NAZARIO, J. The Blaster Worm: Then and Now. *IEEE Security and Privacy Magazine 3*, 4 (2005).

[4] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing 8*, 5 (2011).

[5] BALIGA, A., KAMAT, P., AND IFTODE, L. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *IEEE Symposium on Security and Privacy* (2007).

[6] BENCSÁTH, B., PÉK, G., BUTTYÁN, L., AND FÉLEGYHÁZI, M. The Cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet 4* (2012).

[7] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[8] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. SELECT–Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (1975).

[9] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[10] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data Attacks Are Realistic Threats. In *USENIX Security Symposium* (2005).

[11] CUI, A., AND STOLFO, S. J. Defending Embedded Systems with Software Symbiotes. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2011).

[12] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[13] GOUDEY, H. Microsoft Malware Protection Center, Threat Report: Rootkits. Tech. rep., Microsoft Corporation, June 2012. http://www.microsoft.com/en-us/download/confirmation.aspx?id=34797.

[14] HEX-RAYS. IDA Pro, February 2014. https://www.hex-rays.com/products/ida/.

[15] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[16] HUND, R., HOLZ, T., AND FREILING, F. C. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium* (2009).

[17] KING, J. C. Symbolic execution and program testing. In *Communications of the ACM (CACM)* (1976).

[18] LANZI, A., SHARIF, M. I., AND LEE, W. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[19] LEE, H., MOON, H., JANG, D., KIM, K., LEE, J., PAEK, Y., AND KANG, B. B. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security Symposium* (2013).

[20] LI, J., WANG, Z., BLETSCH, T., SRINIVASAN, D., GRACE, M., AND JIANG, X. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security 6*, 4 (2011).

[21] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Symposium on Network and Distributed System Security (NDSS)* (2011).

[22] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium* (2008).

[23] MICROSOFT-RESEARCH. Z3: Theorem Prover, February 2014. http://z3.codeplex.com/.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007). http://www.valgrind.org/.

[25] PASAREANU, C. S., AND VISSER, W. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf. 11*, 4 (2009).

[26] PETRONI, JR., N. L., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *USENIX Security Symposium* (2006).

[27] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[28] PRAKASH, A., VENKATARAMANI, E., YIN, H., AND LIN, Z. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Conference on Dependable Systems and Networks (DSN)* (Jun 2013).

[29] RAVI, S., RAGHUNATHAN, A., KOCHER, P., AND HATTANGADY, S. Security in embedded systems. *ACM Transactions on Embedded Computing Systems 3*, 3 (2004).

[30] RHEE, J., RILEY, R., XU, D., AND JIANG, X. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In *Availability, Reliability and Security (ARES)* (2009).

[31] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[32] RUSSINOVICH, M., SOLOMON, D. A., AND IONESCU, A. *Windows Internals Part I*, 6 ed. Microsoft Press, 2012.

[33] SCHNEIDER, C., PFOH, J., AND ECKERT, C. Bridging the Semantic Gap Through Static Code Analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security* (2012).

[34] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy* (2010).

[35] SHACHAM, H. The geometry of innocent flesh on the bone. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[36] SHOSHITAISHVILI, Y. Python bindings for Valgrind's VEX IR, February 2014. https://github.com/zardus/pyvex.

[37] SOLAR DESIGNER. Getting around non-executable stack (and fix), Aug. 1997.

[38] SPARKS, S., AND BUTLER, J. Shadow Walker: Raising The Bar For Windows Rootkit Detection. *Phrack 11*, 63 (2005).

[39] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy* (2013).

[40] TIP, F. A survey of program slicing techniques. Tech. rep., Amsterdam, The Netherlands, The Netherlands, 1994.

[41] VOGL, S., PFOH, J., KITTEL, T., AND ECKERT, C. Persistent data-only malware: Function Hooks without Code. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[42] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy* (2010).

[43] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[44] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[45] WEISER, M. Program slicing. In *International Conference on Software Engineering (ICSE)* (1981).

[46] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Conference on Dependable Systems and Networks (DSN)* (2012).

[47] YIN, H., LIANG, Z., AND SONG, D. Hookfinder: Identifying and understanding malware hooking behaviors. In *Symposium on Network and Distributed System Security (NDSS)* (2008).

[48] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium* (2013).

X-Force: Force-Executing Binary Programs for Security Applications

Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu
Purdue University
{*pengf, deng14, xyzhang, dxu*}*@cs.purdue.edu*

Zhiqiang Lin
UT Dallas
zhiqiang.lin@utdallas.edu

Zhendong Su
UC Davis
su@cs.ucdavis.edu

Abstract

This paper introduces X-Force, a novel binary analysis engine. Given a potentially malicious binary executable, X-Force can force the binary to execute requiring no inputs or proper environment. It also explores different execution paths inside the binary by systematically forcing the branch outcomes of a very small set of conditional control transfer instructions. X-Force features a crash-free execution model that can detect and recover from exceptions. In particular, it can fix invalid memory accesses by allocating memory on-demand and setting the offending pointers to the allocated memory. We have applied X-Force to three security applications. The first is to construct control flow graphs and call graphs for stripped binaries. The second is to expose hidden behaviors of malware, including packed and obfuscated APT malware. X-Force is able to reveal hidden malicious behaviors that had been missed by manual inspection. In the third application, X-Force substantially improves analysis coverage in dynamic type reconstruction for stripped binaries.

1 Introduction

Binary analysis has many security applications. For example, given an unknown, potentially malicious executable, binary analysis helps construct its human inspectable representations such as control flow graph (CFG) and call graph (CG), with which security analysts can study its behavior [40, 23, 50, 46, 6, 33]. Binary analysis also helps identify and patch security vulnerabilities in COTS binaries [10, 14, 31, 51, 11]. Valuable information can be reverse-engineered from executables through binary analyses. Such information includes network protocols [44, 12, 7, 47, 28, 32], input formats [27, 29, 13], variable types, and data structure definitions [30, 25, 39]. They can support network sniffing, exploit generation, VM introspection, and forensic analysis.

Existing binary analysis can be roughly classified into static, dynamic, and symbolic (concolic) analysis. Static analysis analyzes an executable directly without executing it; dynamic analysis acquires analysis results by executing the subject binary; symbolic (concolic) analysis is able to generate inputs to explore different paths of a binary. These different styles of analyses have their respective strengths and limitations. Static analysis has difficulty in handling packed and obfuscated binaries. Memory disambiguation and indirect jump/call target analysis are known to be very challenging for static analysis.

Dynamic binary analysis is based on executing the binary on a set of inputs. It is widely used in analyzing malware. However, dynamic analysis is incomplete by nature. The quality of analysis results heavily relies on coverage of the test inputs. Moreover, modern malware [16, 26, 19] has become highly sophisticated, posing many new challenges for binary analysis: (1) For a zero-day binary malware, we typically do not have any knowledge about it, especially the nature of its input, making traditional execution-based analysis [15, 50, 4, 43, 49] difficult; (2) Malware binaries are increasingly equipped with anti-analysis logic [37, 5, 17, 18, 35] and hence may refuse to run even if given valid input; (3) Malware binaries may contain multi-staged, condition-guarded, and environment-specific malicious payloads, making it difficult to reveal all payloads, even if one manages to execute them.

Symbolic [8] and concolic analysis [38, 20, 40, 10] has seen much progress in recent years. Some handle binary programs [40, 10, 33, 6] and can explore various paths in a binary. However, difficulties exist when scaling them to complex, real-world binaries, as they operate by modeling individual instructions as symbolic constraints and using SMT/SAT solvers to resolve the generated constraints. Despite recent impressive progress, SMT/SAT remains expensive. While symbolic and concrete executions can be performed simultaneously so that concrete execution may help when symbolic analysis encounters difficulties, the user needs to provide concrete inputs, called *seed inputs*, and the quality of seed inputs is critical to the execution paths that can be explored. With no or little knowledge about malware input, creating such seed inputs is difficult. Moreover, many existing techniques cannot handle obfuscated or self-modifying binaries.

In this paper, we propose a new, practical execution engine called X-Force. The core enabling technique behind X-Force is *forced execution* which, as its name suggests, *forces an arbitrary binary to execute along different paths without any input or environment setup.* More specifically, X-Force monitors the execution of a binary through dynamic binary instrumentation, systematically forcing a small set of instructions that may affect the execution path (e.g., predicates and jump table ac-

cesses) to have specific values, regardless of their computed values, and supplying random values when inputs are needed. As such, the *concrete* program state of the binary can be systematically explored. For instance, a packed/obfuscated malware can be forced to unpack/deobfuscate itself by setting the branch outcomes of self-protection checks, which terminate execution in the presence of debugger or virtual machine. X-Force is able to tolerate invalid memory accesses by performing on-demand memory allocations. Furthermore, by exploring the reachable state of a binary, X-Force is able to explore different aspects or stages of the binary behavior. For example, we can expose malware's data exfiltration operations, without the presence of the real data asset being targeted.

Compared to manual inspection and static analysis, X-Force is more accurate as many difficulties for static analysis, such as handling indirect jumps/calls and obfuscated/packed code, can be substantially mitigated by the concrete execution of X-Force. Compared to symbolic/concolic analysis, X-Force trades precision slightly for *practicality* and *extensibility*. Note that X-Force may explore infeasible paths as it forces predicate outcomes; whereas symbolic analysis attempts to respect path feasibility through constraint solving[1]. The essence of X-Force will be discussed later in Section 6. Furthermore, executions in X-Force are all concrete. Without the need for modeling and solving constraints, X-Force is more likely to scale to large programs and long executions. The concrete execution of X-Force makes it suitable for analyzing packed and obfuscated binaries. It also makes it easy to port existing dynamic analysis to X-Force to leverage the large number of executions, which will mitigate the incompleteness of dynamic analyses.

Our main contributions are summarized as follows:

- We propose X-Force, a system that can force a binary to execute requiring no inputs or any environment setup.
- We develop a crash-free execution model that could detect and recover from exceptions properly. We have also developed various execution path exploration algorithms.
- We have overcome a large number of technical challenges in making the technique work on real world binaries including packed and obfuscated malware binaries.
- We have developed three applications of X-Force. The first is to construct CFG and CG of stripped binaries, featuring high quality indirect jump and call target identification; the second is to study hidden behavior of advanced malwares; the third one is to apply X-Force in reverse engineering variable types and data structure definitions of executables. Our results show that X-Force substantially advances the state-of-the-arts.

[1]However, due to the difficulty of precisely modeling program behavior, even state-of-the-art symbolic analysis techniques [8, 10, 40] cannot guarantee soundness.

2 Motivation Example

Consider the snippet in Figure 1. It shows a hidden malicious payload that hijacks the name resolution for a specific domain (line 14), which varies according to the current date (in function *genName()*). In particular, it receives some integer input at line 2. If the input satisfies condition *C* at line 3, a *DNSentry* object will be allocated. In lines 5-8, if the input has the *CODE_RED* bit set, it populates the object by calling *genName()* and stores the input and the generated name as a (key, value) pair into a hash table. In lines 12-14, the pair is retrieved and used to guide domain name redirection. Note that the hash table is used as a general storage for objects of various types. In line 10, an irrelevant object *o* is also inserted into the table.

This example illustrates some of the challenges faced by both static and symbolic/concolic analysis. *In static analysis*, it is difficult to determine that the object retrieved at line 12 is the one inserted at line 7 because the abstract domain has to precisely model the behavior of the hash table put/get operations and the condition that `y==x`, which requires context-sensitive and path-sensitive analysis, and disambiguating the memory `bucket[i]` and `bucket[i+4]` in *table_get()* and *table_put()*. The approximations made by many static analysis techniques often determine the object at line 12 could be the one put at line 7 or 10. Performed solely at the binary level, such an analysis is actually much more challenging than described here. *In symbolic/concolic analysis*, one can model the input at line 2 as a symbolic variable such that, by solving the symbolic constraints corresponding to path conditions, the hidden payload might be reached. However, the dictionary read at line 21 will be difficult to handle if the file is *unavailable*. Modeling the file as symbolic often causes scalability issues if it has nontrivial format and size, because the generated symbolic constraints are often complex and the search space for acquiring syntactically correct inputs may be extremely large.

In X-Force, the binary is first executed as usual by providing random inputs. Note that X-Force does not need to know the input format *a priori* as its exception recovery mechanism prevents any crashes/exceptions. In other words, the supply of random input values is merely to allow the execution to proceed, not to drive the execution along different paths. In the first normal run, assume that the false branches of the conditionals at lines 3, 5 and 13 are taken, yielding an uninteresting execu-

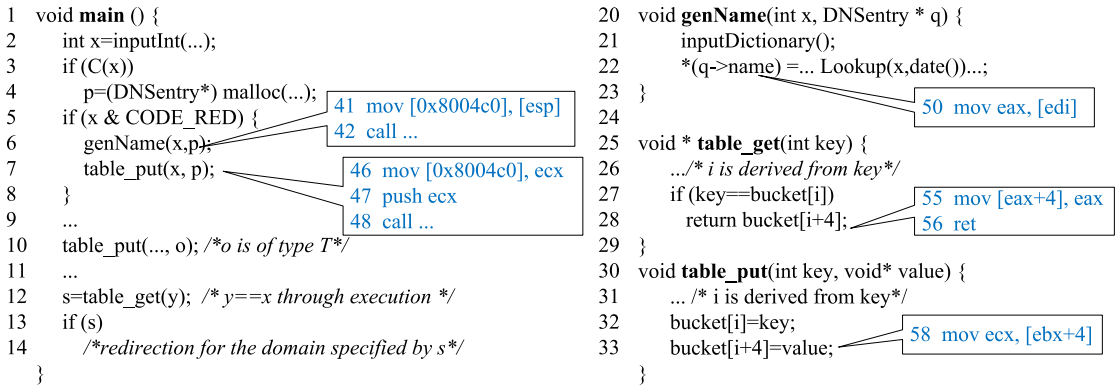


Figure 1: Motivating Example.

tion. X-Force will then try to *force-set* branch outcomes at a small number (say, 1 or 2) of predicates by performing systematic search. Assume that the branch outcome at line 5 is force-set to "true". The malicious payload will be forced to activate. Note that pointer p has a null value at line 6, which will normally crash the execution at line 22. X-Force tolerates such invalid accesses by *allocating memory on demand*, right before line 22. Also, even if the dictionary file at line 21 is absent, X-Force will force it through by supplying random input values. As such, some random integer and domain are inserted into the table (line 7) and retrieved later (line 12). Eventually, the random domain name is redirected at line 14, exposing the DNS hijacking operation. We argue that the domain name itself is not important as long as the hidden hijacking logic is exposed.

3 High Level Design

3.1 Forced Execution Semantics

This section explains the basics of how a single forced execution proceeds. The goal is to have a non-crashable execution. For readability, we focus on explaining how to detect and recover from memory errors in this subsection, and then gradually introduce the other aspects of forced execution such as path exploration and handling libraries and threads in later sections.

Program	P	$::= s$
Stmt	s	$::= s_1; s_2 \mid \texttt{nop} \mid r :=^{\ell} e \mid r :=^{\ell} \texttt{R}(r_a) \mid \texttt{W}^{\ell}(r_a, r_v) \mid \texttt{jmp}^{\ell}(\ell_1) \mid \texttt{if}\ (r^{\ell})\ \texttt{then jmp}(\ell_1) \mid \texttt{jmp}^{\ell}(r) \mid r := \texttt{malloc}^{\ell}(r_s) \mid \texttt{free}^{\ell}(r) \mid \texttt{call}^{\ell}(\ell_1) \mid \texttt{call}^{\ell}(r) \mid \texttt{ret}^{\ell}$
Operator	op	$::= + \mid - \mid * \mid / \mid > \mid < \mid \ldots$
Expr	e	$::= c \mid a \mid r_1\ \texttt{op}\ r_2$
Register	r	$::= \{esp, eax, ebx, \ldots\}$
Const	c	$::= \{true, false, 0, 1, 2, \ldots\}$
Addr	a	$::= \{0, MIN\_ADDR, MIN\_ADDR+1, \ldots, MAX\_ADDR\}$
PC	ℓ	$::= \{\ell_1, \ell_2, \ell_3, \ldots\}$

Figure 2: Language.

Language. Due to the complexity of the x86 instruction set, we introduce a simple low-level language that models x86 binary executables to facilitate discussion. We only model a subset that is sufficient to illustrate the key ideas. Fig. 2 shows the syntax.

Memory reads and writes are modeled by $\texttt{R}(r_a)$ and $\texttt{W}(r_a, r_v)$ with r_a holding the address and r_v the value. Since it is a low-level language, we do not model conditional or loop statements, but rather guarded jumps; `malloc()` and `free()` represent heap allocation and deallocation. Function invocations and returns are modeled by `call()` and `ret`. In our language, stack/heap memory addresses are modeled as a range of integers and a special value 0 to denote the null pointer value. Program counters (or instruction addresses) are explicitly modeled by the *PC* set. Observe that each instruction is labeled with a *PC*, denoting its instruction address. Direct jumps/calls are parameterized with explicit *PC* values whereas indirect jumps/calls are parameterized with a register.

LSet	::=	$\mathcal{P}(Addr)$
$SR \in RegLinearSet$	::=	$Register \mapsto \&LSet$
$SM \in MemLinearSet$	::=	$Addr \mapsto \&LSet$
$accessible \in AddrAccessible$	::=	$Addr \mapsto boolean$

recovery (r) ::=

1: $S \leftarrow SM(r)$
2: $VS \leftarrow \{\}$
3: **for** each address $a \in S$ **do**
4: $\quad VS \leftarrow VS + \{*(a)\}$
5: **end for**
6: $min \leftarrow$ the minimal value in VS
7: $max \leftarrow$ the maximum value in VS
8: $t \leftarrow malloc(max - min + BLOCKSIZE)$
9: $accessible[t, t + max - min + BLOCKSIZE - 1] = true$
10: **for** each address $a \in S$ **do**
11: $\quad offset \leftarrow *(a) - min$
12: $\quad *(a) \leftarrow t + offset$
13: **end for**

Figure 3: Definitions.

In X-Force, we ensure that an execution is not crash-

Table 1: Linear Set Computation Rules.

Statement	Action[1,2]	Rule
initially	`foreach` (global address t) `if` ($isAddr(*t)$) $SM(t) = \{t\}$;	L-INIT
$r := \mathtt{R}(r_a)$	$SR(\text{“}r\text{”}) \rightarrow nil$; `if`($SM(r_a)$) $SR(\text{“}r\text{”}) \rightarrow SM(r_a)$;	L-READ
$\mathtt{W}(r_a, r_v)$	`if` ($SM(r_a)$) $SM(r_a) = SM(r_a) - \{r_a\}$ $SM(r_a) \rightarrow nil$; `if` ($SR(\text{“}r_v\text{”})$) $SR(\text{“}r_v\text{”}) = SR(\text{“}r_v\text{”}) \cup \{r_a\}$; $SM(r_a) \rightarrow SR(\text{“}r_v\text{”})$;	L-WRITE
$r := a$	$SR(\text{“}r\text{”}) \rightarrow \{\}$	L-ADDR
$r := c$ /*!$isAddr(c)$*/	$SR(\text{“}r\text{”}) \rightarrow nil$	L-CONST
$r := r_1 + / - r_2$	`if` (!($isAddr(r_1)$&&$isAddr(r_2)$)) $SR(\text{“}r\text{”}) \rightarrow nil$ `if` ($isAddr(r_1)$) $SR(\text{“}r\text{”}) \rightarrow SR(\text{“}r_1\text{”})$; `if` ($isAddr(r_2)$) $SR(\text{“}r\text{”}) \rightarrow SR(\text{“}r_2\text{”})$;	L-LINEAR
$r := r_1 * / ... r_2$	$SR(\text{“}r\text{”}) \rightarrow nil$	L-NON-LNR
`free`(r)	$t = r$; `while` ($accessible(t)$) `if` ($SM(t)$) $SM(t) = SM(t) - \{t\}$; $t++$;	L-FREE

1. The occurrence“r” denotes the symbolic name of register r, the occurrence of r denotes the value stored in r.
2. Operator “=” means set update, “$\rightarrow$” means pointer update.

able by allocating memory on-demand. However, when we replace a pointer pointing to an invalid address *a* with the allocated memory, *we need to update all the other pointer variables that have the same address value or a value denoting an offset from the address*. We achieve this through the *linear set tracing semantics*, which is also the basic semantics for forced executions[2]. Its goal is to identify the set of variables (i.e. memory locations and registers at the binary level), whose values have *linear correlations*. In this paper, *we say two variables are linearly correlated if the value of one variable is computed from the value of the other variable by adding or subtracting a value.* Note that it is simpler than the traditional definition of linear correlation, which also allows a scaling multiplier. It is however sufficient in this work as *the goal of linear set tracing is to identify correlated pointer variables, which are induced by address offsettings that are exclusively additions and subtractions*.

The semantics is presented in Table 1. The corresponding definitions are presented in Fig 3. Particularly, linear set *LSet* denotes a set of addresses such that *the values stored in these addresses are linearly correlated*. Mapping *SR* maps a register to the reference of a *LSet*. Intuitively, one could interpret that it maps a register to a pointer pointing to a set of addresses such that the values stored in the register and those addresses are linearly correlated. Two registers map to the same reference (of a *LSet*) implies that the values of the two registers are also linearly correlated. Similarly, mapping *SM* maps an address to the reference of a *LSet* such that the values in the address and all the addresses in *LSet* are linearly correlated. The essence of linear set tracing is to maintain the *SR* and *SM* mappings for all registers and addresses that have been accessed so that at any execution point, we can query the set of linearly correlated variables of any given variable.

[2]We will explain the predicate switching part of the semantics in Section 3.2

Table 2: Memory Error Prevention and Recovery.

Statement	Action	Rule
$r := \mathtt{malloc}(r_1)$	`for` ($i = r$ `to` $r + r_1 - 1$) $accessible(i) = true$	M-ALLOC
`free`(r)	$t = r$; `while` ($accessible(t)$) $accessible(t) = false$ $t++$;	M-FREE
$r := \mathtt{R}(r_a)$	`if` (!$accessible(r_a)$) $recovery(r_a)$;	M-READ
$\mathtt{W}(r_a, r_v)$	`if` (!$accessible(r_a)$) $recovery(r_a)$;	M-WRITE

Before execution, the *SM* mapping of all global variables that have an address value is set to the address itself, meaning the variable is only linearly correlated with itself initially (rule L-INIT). Function *isAddr(v)* determines if a value *v* could be an address. X-Force monitors all memory allocations and the image loading process. Thus, given a value, X-Force treats it as a pointer if it falls into static, heap, or stack memory regions. Note that we do not need to be sure that the value is indeed an address. Over-approximations only cause some additional linear set tracing. For a memory read operation, the *SR* mapping of the destination register points to the *SM* set of the value in the address register if the *SM* set exists, which implies the value is an address, otherwise it is set to nil (rule L-READ). Note that in the rule we use “r” to denote the symbolic name of r and r_a to denote the value stored in r_a. $SR(\text{“}r\text{”}) \rightarrow SM(r_a)$ means that we set $SR(\text{“}r\text{”})$ to point to the $SM(r_a)$ set. For a memory write, we first eliminate the destination address from its linear set. Then, the address is added to the linear set of the value register as the address essentially denotes a new linearly correlated variable. Finally, the *SM* mapping of the address is updated (rule L-WRITE). Note that operation “=” means set update, which is different from “$\rightarrow$” meaning set reference update. For a simple address assignment, the *SR* set is set to pointing to an empty linear set, which is different from a nil value (rule L-ADDR). The empty set is essentially an *LSet* object that could be pointed to by multiple registers to denote their linear correlation. A nil value cannot serve this purpose. For a linear operator, the *SR* mapping of the destination register is set to pointing to the *SR* mapping of the register holding an address value (rule L-LINEAR). Intuitively, this is because we are only interested in the linear correlation between address values (for the purpose of memory error recovery). For heap de-allocation, we have to remove each de-allocated address from its linear set (rule L-FREE).

Table 2 presents the set of memory error detection and

recovery rules. The relevant definitions are in Fig. 3. An auxiliary mapping *accessible*() is introduced to denote if an address has been allocated and hence accessible. The M-ALLOC and M-FREE rules are standard. Upon reading or writing an un-accessible address, X-Force calls function *recovery ()* with the register holding the invalid address to perform recovery. In the function, we first acquire the values of all the variables in the linear set and identify the minimal and maximum values (lines 1-6). Note that the values may be different (through address offsetting operations). We then allocate a piece of memory on demand according to the range of the values and a pre-defined default memory block size. Then in lines 9-12, the variables in the linear set are updated according to their offsets in the block. We want to point out that on-demand allocation may not allocate enough space. However, such insufficiency will be detected when out-of-bound accesses occur and further on-demand re-allocation will be performed. We also want to point out that a correctly developed program would first write to an address before it reads. As such, the on-demand allocation is often triggered by the first write to an invalid buffer such that the value could be correctly written and later read. In other words, we do not need to recover values in the on-demand allocated buffers.

In our real implementation, we also update all the registers that are linearly correlated, which can be determined by identifying the registers pointing to the same set. Furthermore, the rules only describe how we ensure heap memory safety whereas X-Force protects critical stack addresses such as return addresses and parameters, which we will discuss later.

Example. Fig. 4 presents part of a sample execution with the linear set tracing and memory safety semantics. The program is from the motivation example (Fig. 1). In the execution, the else branch of line 3 is taken but the true branch of line 5 is forced. As such, pointer p has a null value when it is passed to function *genName()*, which would cause an exception at line 22. In Fig. 4, we focus on the executions of lines 6, 22 and 7. The second column shows the binary code (in our simplified language). The third column shows the corresponding linear set computation and memory exception detection and recovery. Initially, $SM(\&p = 0x8004c0)$ is set to pointing to the set $\{0x8004c0\}$ according to rule L-INIT. At binary code line 2, $SR(eax)$ is set to pointing to the set of $SM(\&p)$. At line 3, since the value is further copied to a stack address $0xce0080$, *eax*, $\&p$ and the stack address all point to the same linear set containing $\&p$ and the stack address. Intuitively, these are the three variables that are linearly correlated. At lines 9 and 10, *edi* further points to the same linear set. At line 12, when the program tries to access the address denoted by $edi = 4$, the memory safety component detects the exception and performs on demand allocation. According to the linear set, $\&p$ and the stack address $0xce0080$ are set to the newly allocated address $0xd34780$ while *edi* is updated to $0xd34784$ according to its offset. While it is not presented in the table, the program further initializes the newly allocated data structure. As a result, when pointer p is later passed to *table_put()*, it points to a valid data structure. □

Algorithm 1 Path Exploration Algorithm

Output:	Ex - the set of executions (each denoted by a sequence of switched predicates) achieving a certain given goal (e.g. maximum coverage)
Definition	*switches*: the set of switched predicates in a forced execution, denoted by a sequence of integers. For example, $1 \cdot 3 \cdot 5$ means that the 1st, 3rd, and 5th predicates are switched $WL : \mathcal{P}(\overline{Int})$ - a set of forced executions, each denoted by a sequence of switched predicates $preds : \overline{Predicate \times boolean}$ - the sequence of executed predicates with their branch outcomes

1: $WL \leftarrow \{\text{nil}\}$
2: $Ex \leftarrow$ nil
3: **while** WL **do**
4: $switches \leftarrow WL$.pop()
5: $Ex \leftarrow Ex \cup switches$
6: Execute the program and switch branch outcomes according to *switches*, update fitness function $\mathcal{F}$
7: $preds \leftarrow$ the sequence of executed predicates
8: $t \leftarrow$ the last integer in *switches*
9: $preds \leftarrow$ remove the first t elements in $preds$
10: **for** each $(p,b) \in preds$ **do**
11: **if** $eval(\mathcal{F},p,b)$ **then**
12: update fitness function $\mathcal{F}$
13: $WL \leftarrow WL \cup switches \cdot t$
14: **end if**
15: $t \leftarrow t+1$
16: **end for**
17: **end while**

In the early stage of the project, we tried a much simpler strategy that is to terminate a forced execution when an exception is observed. However, we observed that since we do not provide any real inputs, exceptions are very common. Furthermore, simply skipping instructions that cause exceptions did not work either because that would have cascading effects on program state corruption. Finally, a crash-proof execution model as proposed turned out to be the most effective one.

X-Force also automatically recovers from other exceptions such as division-by-zero, by skipping those instructions that cause exceptions. Details are omitted.

Source Code Trace	Binary Code Trace	Linear Set Computation and Memory Safety
6 genName(x,p);	1. ebx= 0x8004c0;	**SR(ebx)**→ {}
	2. eax= R(ebx); /* eax=0 */	**SR(eax)**, SM(0x8004c0) → {0x8004c0}
	3. W(esp, eax); /* esp=0xce0080 */	**SM(0xce0080)**, SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	4. …	
	5. call genName;	
/* in ***genName***(... DNSentry * q) */	6. ...	
22 *(q->name) =... Lookup(...)	7. ebx= ebp;	
	8. ebx= ebx + 8; /* ebx=0xce0080*/	
	9. edi= R(ebx);	**SR(edi)**, SM(0xce0080), SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	10. edi =edi + 4; /* edi= 0+4=4*/	**SR(edi)**, SM(0xce0080), SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	11. eax=…; /* eax=Lookup(...) */	SR(edi),SM(0xce0080),SM(0x8004c0)→ {0x8004c0, 0xce0080}
	12. W(edi, eax);	Exception! *0xce0080 = *0x8004c0 = malloc (4+BLOCKSIZE) = 0xd34780 edi= 0xd34780 + 4=0xd34784
7 table_put(x,p);	13. …	
	14. ebx= 0x8004c0;	
	15. ecx= R(ebx);	ecx= 0xd34780
	16. W(esp, ecx); /* esp=0xce0080 */	
	17. call table_put;	

Figure 4: Sample Execution for Linear Set Tracing and Memory Safety. The code is from Fig. 1.

3.2 Path Exploration in X-Force

One important functionality of X-Force is the capability of exploring different execution paths of a given binary to expose its behavior and acquire complete analysis results. In this subsection, we explain the path exploration algorithm and strategies.

To simplify discussion, we first assume a binary only performs control transfer through simple predicates (i.e. predicates with constant control transfer targets). We will introduce how to extend the algorithms in realistic settings, e.g., supporting exploration of indirect jumps in later section.

Algorithm 1 describes a general path exploration algorithm, which generates a pool of forced executions that are supposed to meet our goal specified by a configurable fitness function. It is a work list algorithm. The work list stores a list of (forced) executions that may be further explored by switching more predicates. Each execution is denoted by a sequence of integer numbers that specify the executed predicate instances to switch. Note that X-Force only force-sets the branch outcome of a small set of predicate instances. It lets the other predicate instances run as usual. Initially (line 1), the work list is a singleton set with a nil sequence, representing an execution without switching any predicate. Note that the work list is not empty initially. At the end of a forced execution, we update the fitness function that indicates the remaining space to explore (line 6), e.g., coverage. Then in lines 7-16, we try to determine if it would be of interest to further switch more predicate instances. Lines 7-9 compute the sequence of predicate instances eligible for switching. Note that it cannot be a predicate before the last switched predicate specified in *switches* as switching such a predicate may change the control flow such that the specification in *switches* becomes invalid. In lines 10-16, for each eligible predicate and its current branch outcome, we query the fitness function to determine if we should further switch it to generate a new forced execution. If so, we add it to the work list and update the fitness function. Note that in each new forced execution, we essentially switch one more predicate.

Different Fitness Functions. The search space of all possible paths is usually prohibitively large for real-world binaries. Different applications may define different fitness functions to control the scope they want to explore. In the following, we introduce three fitness functions that we use. Other more complex functions can be similarly developed.

- *Linear Search.* In certain applications, such as constructing control flow graphs and dynamic type reverse engineering (Section 5), the goal may be just to cover each instruction. The fitness function $\mathcal{F}$ could be defined as a mapping $covered : Predicate \times boolean \mapsto boolean$ that determines if a branch of a predicate has been covered. The evaluation in the box in line 11 of Algorithm 1 is hence defined as $!covered(p, \neg b)$, which means we will switch the predicate if the other branch has not been covered. Once we decide to switch an additional predicate, the fitness function is updated to reflect the new coverage (line 12). The number of executions needed is hence $O(n)$ with n the number instructions in the binary.
- *Quadratic Search.* In applications such as identifying indirect call targets, which is a very important challenge in binary analysis, simply covering all instructions may not be sufficient, we may need to cover paths that may lead to indirect calls or generate different indirect call targets. We hence define $\mathcal{F}$ as a set *icalls* to keep the set of the indirect call sites and potential indirect call targets that have been discovered by all the explored paths. The evaluation in line 11 is hence to test if cardinality of *icall* grows with the currently explored path. If so, the execution is considered important and all eligible unique predicates (not instances) in the execution are further explored. The complexity is $O(n^2)$ with n the number of instructions. X-Force can also limit the

quadratic search within a function.

- *Exponential Search.* If we simply set the evaluation in the line 12 to true, the algorithm performs exponential search because it will explore each possible combination. In practice, we cannot afford such search. However, X-Force provides the capability for the user to limit such exponential search within a sub-range of the binary.

Taint Analysis to Reduce Search Space. An observation is that we do not have to force-set predicates in low-level utility methods, because their branch outcomes are usually not affected by any input. Hence in X-Force, we use taint analysis to track if a predicate is related to program input. X-Force will only force branch outcomes of those tainted predicates. Since this is a standard technique, we omit its details.

4 Practical Challenges

In this section, we discuss how we address some prominent challenges in handling real world executables.

Jump Tables. In our previous discussion, we assume control transfer is only through simple predicates. In reality, jump tables allow a jump instruction to have more than two branches. Jump tables are widely used. They are usually generated from `switch` statements in the source code level. In X-Force, we leverage existing jump table reverse engineering techniques [21] to recover the jump table for each indirect jump. Our exploration algorithm then tries to explore all possible targets in the table.

Handling Loops and Recursions. Since X-Force may corrupt variables, if a loop bound or loop index is corrupted, an (incorrect) infinite loop may result. Similarly, if X-Force forces the predicate that guards the termination of some recursive function call, infinite recursion may result. To handle infinite loops, X-Force leverages taint analysis to determine if a loop bound or loop index is computed from input. If so, it resets the loop bound/index value to a pre-defined constant. To handle infinite recursion, X-Force constantly monitors the call stack. If the stack becomes too deep, X-Force further checks if there are cyclic call paths within the call stack. If cyclic paths are detected, X-Force skips calling into that function by simulating a "ret" instruction.

Protecting Stack Memory. Our early discussion on memory safety focused on protecting heap memory. However, it is equally important to protect stack memory. Particularly, the return address of a function invocation and the stack frame base address of the caller are stored on stack upon the invocation. They are restored when the callee returns. Since X-Force may corrupt variable values that affect stack accesses, such critical data could be undesirably over-written. We hence need to protect stack memory as well. However, we cannot simply prevent any stack write beyond the current frame. The strategy of X-Force is to prevent any stack writes that originate in the current stack-frame to go beyond the current frame. Specifically, when a stack write attempts to over-write the return address, the write is skipped. Furthermore, the instruction is flagged. Any later instances of the instruction that access a stack address beyond the current stack-frame are also skipped. The flags are cleared when the callee returns.

Handling Library Function Calls. The default strategy of X-Force is to avoid switching predicates inside library calls as our interest falls in user code. X-Force handles the following library functions in some special ways.

- *I/O functions.* X-Force skips all output calls and most input calls except file inputs. X-Force provides wrappers for file opens and file reads. If the file to open does not exist, X-Force skips calling the real file open and returns a special file handler. Upon file reads, if the file handler has the special value, it returns without reading the file such that the input buffer contains random values. Supporting file reads allows X-Force to avoid unnecessary failure recovery and path exploration if the demanded files are available.
- *Memory manipulation functions.* To support memory safety, X-Force wraps memory allocation and de-allocation. For memory copy functions such as `memcpy()` and `strcpy()`, the X-Force wrappers first determine the validity of the copy operation, e.g., the source and target address ranges must have been allocated, must not overlap with any critical stack addresses. If necessary, on-demand allocation is performed before calling the real function. This eliminates the need of memory safety monitoring, linear set tracing, and memory error recovery inside these functions, which could be quite heavy-weight due to the special structure of these functions. For example, `memcpy()` copies individual addresses one by one and these addresses are linearly correlated as they are computed through pointer manipulation, leading to very large linear sets.

For statically linked executables, X-Force relies on IDA-Pro to recognize library functions in a pre-processing step. IDA leverages a large signature dictionary to recognize library functions with very good accuracy. For functions that are not recognized by IDA, X-Force executes them as user code.

Handling Threads. Some programs spawn additional threads during their execution. It is difficult for X-Force to model multiple threads into a single execution since the order of their execution is nondeterministic. If we

simply skip the thread creation library functions such as *CreateThread()* and *beginthread()*, the functions in the thread could not be covered. To solve this problem, we adopt a simple yet effective approach of serializing the execution of threads. The calls to thread creation library functions are replaced with direct function calls to the starting functions of threads, which avoid creating multiple threads and guarantees code coverage at the same time. Note that as a result, X-Force is incapable of analyzing behavior that is sensitive to schedules. We will leave it to our future work.

5 Evaluation

X-Force is implemented in PIN. It supports WIN32 executables. In this section, we use three application case studies to demonstrate the power of X-Force.

Table 4: Detailed Coverage Comparison with Dynamic Analysis

	Input Union	X-Force	Input Union ∩ X-Force	Input Union \ X-Force	X-Force \ Input Union
164.gzip	3601	5075	3601	0	1474
175.vpr	19398	29218	19398	0	9820
176.gcc	157451	227546	157451	0	70095
181.mcf	1622	1935	1622	0	313
186.crafty	27811	42763	27811	0	14952
197.parser	17339	23135	17339	0	5796
252.eon	15580	27224	15580	0	11644
253.perlbmk	55964	33643	27003	28961	6640
254.gap	37564	110066	37564	0	72502
255.vortex	53798	101207	53798	0	47409
256.bzip2	3612	4830	3612	0	1218
300.twolf	19996	41935	19996	0	21939

5.1 Control Flow Graph (CFG) and Call Graph (CG) Construction

Construction of CFG and CG is a basic but highly challenging task for binary analysis, especially the identification of indirect call targets. In the first case study, we apply X-Force to construct CFGs and CGs for stripped SPECINT 2000 binaries. We also evaluate the performance of X-Force in this study. To construct CFGs and CGs, we use X-Force to explore execution paths and record all the instructions, control flow edges, and call edges, including indirect jump and indirect call edges. The exploration algorithm is a combination of linear search and quadratic search (Section 3.2). Quadratic search is limited to functions that contain indirect calls or encounter values that look like function pointers.

We compare X-Force results with four other approaches: (1) IDA-Pro; (2) Execute all the test cases provided in SPEC and union the CFGs and CGs observed for each program (i.e., dynamic analysis); (3) Static CG construction using LLVM on SPEC source code (i.e., static analysis) [3]. (4) Dynamic CFG construction using a symbolic execution system S2E [10]. We could not compare with CodeSurfer-X86 [2], which can also generate CFG/CG for executables based on static analysis, because it is not available through commercial or academic license.

Part of the results is presented in Table 3. Columns 2-4 present the instructions that are covered by the different approaches. Particularly, the second column shows the number of instructions recognized by IDA. The third column shows those that are executed by concrete input runs. Columns 5-8 show the indirect call edges recognized by the different approaches[4]. The last five columns show internal data of X-Force.

From the coverage data, we observe that X-Force could cover a lot more instructions than dynamic analysis except `253.perlbmk`. Note that the dynamic analysis results are acquired using all the test, training and reference inputs in SPEC, which are supposed to provide good coverage. Table 4 presents more detailed coverage comparison with dynamic analysis. Observe that X-Force covers all the instructions that are covered by natural runs for all benchmarks except `253.perlbmk`, which we will explain later. X-Force could cover most of the instructions identified by IDA except `252.eon` and `253.perlbmk`. We have manually inspected the differences between the IDA and X-Force coverage. For most programs except `253.perlbmk`, the differences are caused by part of the code in those binaries being unreachable. In other words, they are dead code that cannot be executed by any input. Since IDA simply scans the code body to construct CFG and CG, it reports all instructions it could find including the unreachable ones.

Table 5: Detailed Indirect Call Edges Identification Comparison with Dynamic Analysis

	Input Union	X-Force	Input Union ∩ X-Force	Input Union \ X-Force	X-Force \ Input Union
164.gzip	2	2	2	0	0
176.gcc	169	1720	169	0	1551
252.eon	60	121	60	0	61
253.perlbmk	225	151	103	122	48
254.gap	1103	20485	1103	0	19382
255.vortex	28	30	28	0	2

Indirect call edge identification is very challenging in binary analysis as a call site may have multiple call targets depending on execution states, which are usually difficult to cover or abstract. Some of them are dependent on states related to multiple procedures. Note that there does not exist an oracle that can provide the ground truth for the set of real indirect call edges. From the results, we could observe that LLVM's indirect call identification algorithm generates a large number of edges, much more than X-Force. However, we confirm that most of them are bogus because the LLVM algorithm simply relies on method signatures to identify possible targets and

[3] We cannot compare LLVM CFGs with X-Force CFGs as LLVM CFGs are not represented at the instruction level.

[4] Direct jump and call edges are easy to identify and elided.

Table 3: CFG and CG Construction Results.

	Coverage			Indirect Call Edge				X-Force Internals				
	IDA-Pro	Input Union	X-Force	IDA-Pro	Input Union	LLVM	X-Force	Time (s)	# of Runs	Avg. # of Exp.	Avg./Max. Linear Set Size	Switched/Total # of predicates
164.gzip	7913	3601	5075	0	2	2	2	704	246	10	2.9/36	2.1/1291
175.vpr	31847	19409	29218	0	0	0	0	8725	1849	49	2.8/19	4.7/2164
176.gcc	310277	157451	227546	25	169	9141	1720	173241	26606	95	4.5/265	12.9/29847
181.mcf	2184	1622	1935	0	0	0	0	129	113	10	3.1/23	4.3/153
186.crafty	43327	27811	42763	0	0	0	0	43995	2496	0.4	2.6/9	8.0/62582
197.parser	25532	17339	23135	0	0	0	0	3424	1820	8	2.5/17	6.4/944
252.eon	70592	15580	27224	0	60	28802	121	6379	2091	4	2.3/10	4.1/3146
253.perlbmk	132264	55964	33643	24	225	-	151	7137	843	0.8	3.5/40	8.3/9535
254.gap	113410	37564	110066	2	1103	187155	20470	50745	7319	1353	30.0/1846	6.0/173316
255.vortex	132053	53798	101207	0	28	340	30	34776	8566	13	2.9/33	7.3/2548
256.bzip2	5761	3612	4830	0	0	0	0	557	209	5	3.3/15	1.4/7001
300.twolf	46556	19996	41935	0	0	0	0	10043	2825	17	2.6/8	5.4/1322

hence is too conservative. X-Force could recognize a lot more indirect call edges than dynamic analysis. The detailed comparison in Table 5 shows that the X-Force results cover all the dynamic results and have many more edges, except *253.perlbmk*. We have manually inspected a random set of the selected edges that are reported by X-Force but not the dynamic analysis and confirmed that they are feasible. From the results in Table 3, IDA can hardly resolve any indirect call edges.

Table 6: Result of using S2E to analyze SPEC programs

	Basic Block Coverage	Function Block Coverage	Touched Functions	Fully Covered Functions	Number of Paths
164.gzip	768/2240(34%)	768/1294(59%)	62/186(33%)	21/186(11%)	134
176.gcc	740/46487(1%)	740/1468(50%)	62/1398(4%)	19/1398(1%)	261
252.eon	64/2830(2%)	64/101(63%)	19/649(2%)	13/649(2%)	33
253.perlbmk	1708/37384(4%)	1708/6912(24%)	134/1510(8%)	27/1510(1%)	329
254.gap	1235/28871(4%)	1235/3136(39%)	80/941(8%)	21/941(2%)	29
255.vortex	10933/35979(30%)	10933/20822(52%)	437/1031(42%)	21/1031(2%)	9

We also use S2E to analyze the six SPECINT 2000 programs that contain indirect calls. The four programs other than `252.eon` and `255.vortex` read input from *stdin*, so we use the *s2ecmd* utility tool provided by S2E to write 64 bytes to *stdout* and pipe the symbolic bytes into these programs. We run each program in S2E and use the `ExecutionTracer` plugin to record the execution trace. We use the IDA scripts provided by S2E to extract information of basic blocks and functions from the binaries, and then use the `coverage` tool provided by S2E to calculate the result.

The result is shown in Table 6. The columns show the following metrics from left to right: (1) coverage of basic blocks; (2) coverage of basic blocks when excluding the basic blocks in those functions that are not executed; (3) coverage of functions; (4) percentage of fully-covered functions; (5) the number of different paths that S2E explored. Observe that the coverage is much lower than X-Force in general. `176.gcc`, `253.perlbmk` and `254.gap` are parsers/compilers. They have poor coverage on S2E because they get stuck in the parsing loops/automatas, whose termination conditions are dependent on the symbolic input. Regarding `255.vortex`, S2E fails to solve the constraints when an indirect jump uses the symbolic variable as the index of jump table. As a result, S2E fails to identify most of the indirect call edges due to the failure of creating different objects. In `252.eon`, S2E fails to solve the constraints of the input file format, which must contain a specific string as header. The program throws exception and terminates quickly, which leads to poor coverage.

`253.perlbmk` is a difficult case for X-Force. It parses perl source code to generate syntax trees. The indirect call targets are stored in the nodes of syntax trees. However, since the syntax tree construction is driven by finite automata, path coverage does not seem to be able to cover enough states in the automata to generate enough syntax trees of various forms. A few other benchmarks such as `176.gcc` and `254.gap` also leverage automata based parsers, however their indirect call targets are not so closely-coupled with the state of the automata and hence X-Force can still get good coverage. We will leave it to our future work to address this problem.

The last five columns show some statistics of X-Force. The run time and the number of explorations are largely linear regarding the number of instructions except for a small number of functions on which quadratic search is performed. Some take a long time (e.g., close to 50 hours for `176.gcc`) due to their complexity. The average number of exceptions is the number of exceptions encountered and recovered from in each execution (e.g. memory exceptions, division by zero). The numbers are smaller than we expected given that we execute these programs without any inputs and switch branch outcomes. It shows that our exception recovery could effectively prevent cascading exceptions. The linear set sizes are manageable. The last column shows the average number of switched predicates versus the average number of predicate instances in total in an execution. It shows that X-Force may violate path feasibility only in a very small part of execution. The performance overhead of X-Force compared to the vanilla PIN is 473 times on average. It is measured by comparing the number of instructions that could be executed by X-Force and the vanilla PIN within the same amount of time.

5.2 Malware Analysis

One common approach to understanding the behavior of an unknown malware sample is by looking at the library calls it makes. This could be done by static, dynamic or symbolic analysis; however, they all have limitations.

Table 7: Result of using X-Force for malware analysis compared with IDA Pro and native run.

Name	MD5	File Size(KB)	Number of Library Functions			Number of Library Call Sites			No. of Runs
			IDA Pro	Native Run	X-Force	IDA Pro	Native Run	X-Force	in X-Force
dg003.exe	4ec0027bef4d7e1786a04d021fa8a67f	192	147	129	252	808	546	1750	800
Win32/PWSteal.F	04eb2e58a145462334f849791bc75d18	20	7	21	42	9	28	94	30
APT1.DAIRY	995442f722cc037885335340fc297ea0	19	90	40	100	213	68	236	121
APT1.GREENCAT	0c5e9f564115bfcbee66377a829de55f	14.5	66	26	64	303	114	302	112
APT1.HELAUTO	47e7f92419eb4b98ff4124c3ca11b738	8.5	41	16	39	109	33	109	30
APT1.STARSYPOUND	1f2eb7b090018d975e6d9b40868c94ca	7	37	14	36	80	15	80	25
APT1.WARP	36cd49ad631e99125a3bb2786e405cea	45.5	77	47	79	495	156	414	221
APT1.NEWSREEL	2c49f47c98203b110799ab622265f4ef	21	67	31	67	189	49	192	93
APT1.GOGGLES	57f98d16ac439a11012860f88db21831	10.5	35	21	36	127	45	131	42
APT1.BOUNCER	6ebd05a02459d3b22a9d4a79b8626bf1	56	11	16	97	24	39	562	298

Static analysis could not obtain the parameters of library calls that are dynamically computed and is infeasible when the sample is packed or obfuscated. Traditional dynamic analysis can obtain parameters and is immune to packing and obfuscation, however, it could only explore some of the execution paths depending on the input and the environment. Unfortunately, the input is usually unknown for malware. Symbolic analysis, while being able to construct input according to path conditions, has difficulty in handling complex or packed binaries.

X-Force overcomes these problems as traditional dynamic analysis could be built upon X-Force to explore various execution paths without providing any inputs or the environment. In this case study, we demonstrate the use of a library call analysis system we built on top of X-Force to analyze real-world malware samples.

When we implement library call analysis on top of X-Force, we slightly adjust X-Force to make it suitable for handling malware: (1) We enable the concrete execution of most library functions including output functions because many packers use output functions (e.g. `RtlDecompressBuffer()`) to unpack code. We continue to skip some library calls such as `Sleep()` and `DeleteFile()`; (2) We intercept a few functions that allocate memory and change page attributes, such as `VirtualAlloc()` and `VirtualProtect()`. This is for tracking the memory areas of code and data which keep changing at runtime due to self-modifying and dynamically generated code.

Given a malware sample, we use X-Force to explore its paths. We use the linear search algorithm (Section 3.2) as it provides a good balance between efficiency and coverage. During each execution, we record a trace of function calls. For library calls, we also record the parameter values. The trace is then transformed into an interprocedural flow graph that has control transfer instructions, including jumps and calls, as its nodes, and control-flow/call edges as its edges. The parameters of library calls are also annotated on the graph. The graphs generated in multiple executions are unioned to produce the final graph. We then manually inspect the final graphs to understand malware behavior.

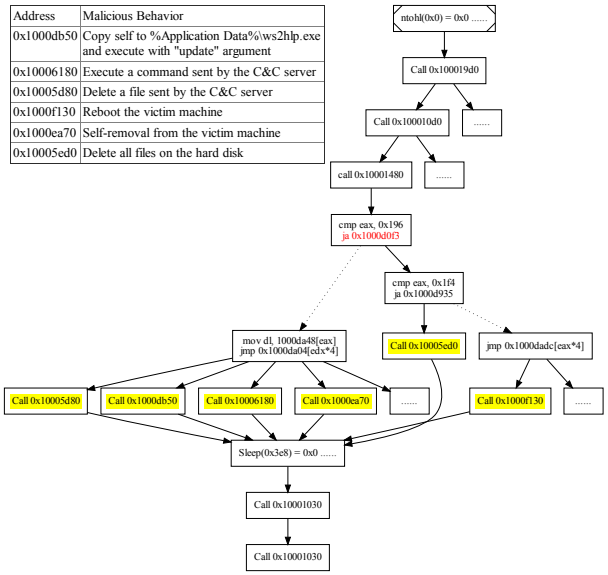


Figure 5: The flow graph of the function at *0x1000c630* generated by X-Force when analyzing dg003.exe.

We evaluate our system on 10 real-world malware samples which are either wild-captured virus/trojan or APT samples described in [9]. Since our analysis focuses on library calls, we choose the number of identified library functions and the total number of their call sites as the evaluation metric[5]. We also compare our results with IDA-Pro and the native run. In IDA, library functions are identified from the import table; the call sites are identified by scanning the disassemblies. In the native run, we execute the malware without any arguments and record the library calls using a PIN tool.

The results are shown in Table 7. We can see that for packed or obfuscated samples such as *dg003.exe*, *Win32/PWSteal.F*, *APT1.DAIRY*, and *APT1.BOUNCER*, IDA gets fewer library functions and call sites compared to X-Force. For other samples that are not packed or obfuscated, since the executables could be properly disassembled, the metrics obtained in IDA and X-Force are

[5]We exclude the C/C++ runtime initialization functions which are only called before the main function.

very close. However, even in such cases, static analysis is insufficient to understand the malicious behavior because it does not show the values of the library function parameters. Compared to the native run method, X-Force can identify more library functions and call sites.

Next, we present detailed analysis for two representative samples.

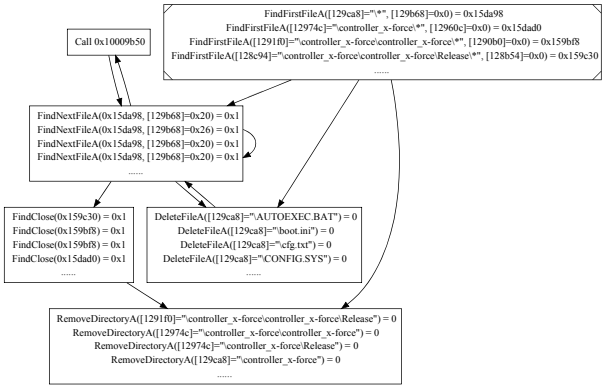


Figure 6: The flow graph of the function at *0x10009b50* in dg003.exe that delete all files on the hard disk.

Dg003.exe. This is a typical APT malware sample that features multi-staged, condition-guarded and environment-specific payload. In the first stage, the malware extracts a DLL which it carries as its resource, packs the DLL in memory using a proprietary algorithm and writes the packed DLL to the disk. In the second stage, the packed DLL is loaded, unpacks itself in memory and executes the main payload.

There is a previous report [26] in which the analysts used both static and dynamic analyses to analyze this sample. To perform static analysis using IDA Pro, they manually extract and unpack the DLL. This requires reverse engineering the unpacking algorithm, which could be both time consuming and difficult. Our system avoids such trouble by concretely executing the unpacking routine which performs the unpacking for us. Compared with their dynamic analysis, it takes X-Force about 5 hours to finish 800 executions to explore all paths in both the first and second stages of the malware. After that, the traces are transformed into a flow graph containing 378 functions. Our system is able to discover a set of malicious behaviors that are *NOT* mentioned in the previous report. As shown in Fig. 5, each highlighted function call in the graph corresponds to a previously unrevealed malicious behavior. Each behavior is identified using the library calls made in the corresponding function. For example, as shown in Fig. 6, the library calls and the parameters in the function at *0x10009b50* show that it recursively enumerates and deletes files and directories starting from the root directory, which indicates its behavior is to delete all files on the disk.

In Fig. 5 we can see that the common dominator of all these function calls (highlighted in red color) determines if the value of *eax* register is larger than 0x196. With taint analysis in X-Force, we find that the value of the *eax* register is related to an input which is a buffer in a previous *recv* library function call. This indicates it represents the command ID sent by the C&C server, which leads to the execution of different malicious behaviors. Hence, we suspect that the previous analysts missed some behaviors because the C&C server only sent part of the possible commands at the time they ran the malware. We also find that the buffer in the *recv* function call is translated to the command ID using a private decryption algorithm, so it would be infeasible for symbolic analysis to solve the constraints and construct a valid input. We also want to point out that at the time we perform the analysis, the C&C server of this malware is already inactive; we would not be able to discover these malicious behaviors, had we not used X-Force.

Win32/PWSteal.F. Before trying X-Force on this sample, we first try static analysis using IDA-Pro. Surprisingly, this sample does not import any suspicious library function; not even a function that could perform I/O (e.g. read/write file, registry or network socket). The `LoadLibrary()` and `GetProcAddress()` functions are not imported either, which means the common approach of dynamically loading libraries is not used. The strings in the executable do not contain any DLL name or library function name either. This indicates the sample is equipped with advanced API obfuscation technique to thwart static analysis.

Since static analysis is infeasible, we submit the sample to the Anubis malware analysis platform for dynamic analysis. The result shows the malware does read some registry entries and files, however, none of them seems malicious. Hence, we feed the sample to our system in hopes of revealing its real intent. X-Force achieves full coverage after exploring 30 paths and generates a graph with 15 functions. By traversing the graph, we find that this malware aims at stealing the password that is stored by IE and Firefox in the victim's machine. It enumerates the registry entry that stores the encrypted auto-complete password for IE and calls library functions such as `CryptUnprotectData()` to decrypt the stored password. This is very similar to the attack mentioned in [1]. Regarding Firefox, it first gets the user name from `profiles.ini` under the Firefox application data directory, and then steals the password that is stored in the `signons*.txt` under the directory of the user name. The password is then uploaded to a remote FTP server using the file name `[Computer Name].[IP Address].txt`. Clearly, this sample finds the entry addresses of these library functions at runtime using some obfuscation techniques. X-Force allows us to identify the

malicious behavior without spending unnecessary time on reverse-engineering the API obfuscation.

Moreover, the flow graph also reveals the reason why Anubis missed the malicious behavior: the malware performs environment checks to make sure the targets exist before trying to attack. For example, in the function where the malware steals password from IE, it will try to open the registry entry that contains the auto-complete password; if such entry does not exist or is empty, the malware will cease its operation and return from that function. Also, before it tries to steal password stored by Firefox, it will first try querying the installation directory of Firefox from registry to make sure the target program exists in the system. Such "prerequisites" are unlikely to be fulfilled in automated analysis systems as they are unpredictable. However, by force-executing through different paths, X-Force is able to get through these checks to reveal the real intent of the malware.

```
TYPE_1 func1(TYPE_2 arg1, TYPE_3 arg2) {
        TYPE_4 var1;
1       var1 = strlen (arg1);
2       if (arg2 >= var1)
3               return 0;
4       return arg1[arg2];
}
```

Figure 7: REWARDS example.

5.3 Type Reverse Engineering

Researchers have proposed techniques to reverse engineer variable and data structure types for stripped binaries [30, 39, 25]. The reverse engineered types can be used in forensic analysis and vulnerability detection. There are two common approaches. REWARDS [30] and HOWARD [39] leverage dynamic analysis. They can produce highly precise results but incompleteness is a prominent limitation – they cannot reverse engineer types of variables if such variables are not covered by executions. TIE [25] leverages static analysis and abstract interpretation such that it provides good coverage. However, it is challenging to apply the technique to large and complex binaries due to the cost of analysis.

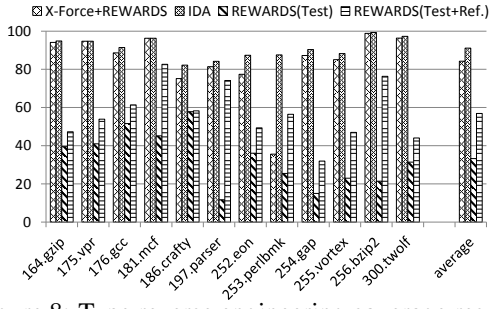


Figure 8: Type reverse engineering coverage results.

One advantage of X-Force is that the forced executions are essentially concrete executions such that existing dynamic analyses could be easily ported to X-Force to benefit from the good coverage. Therefore in the third case study, we port the implementation of REWARDS to X-Force. Given a binary executable and a few test inputs, REWARDS executes it while monitoring dataflow during execution. When execution reaches system or library calls, the types of the parameters of these calls are known. Such execution points are called *type sinks*. Through the dynamic dataflow during execution, such types could be propagated to variables that (transitively) contributed to the parameters in the past and to variables that are (transitively) dependent on these parameters.

Consider the example in Fig. 7. Assume `func1` is executed. After line 1, the type of `arg1` and `var1` get resolved using the interface of `strlen()`. So `TYPE_2` is `char *`, and `TYPE_4` is `unsigned int`. In line 2, `arg2` is compared with `var1`, implying they have the same type. Thus `TYPE_3` gets resolved as `unsigned int`. Later when line 4 gets executed, it returns `TYPE_1` which is resolved as `char` since `arg1` is of `char *`.

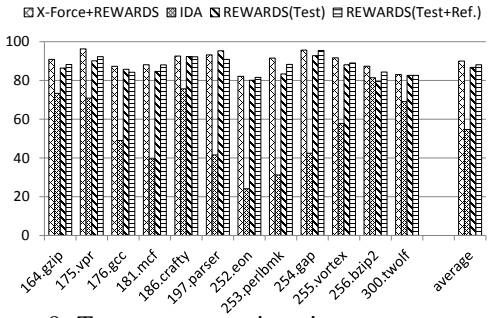


Figure 9: Type reverse engineering accuracy results.

Porting REWARDS to X-Force requires very little modification of either the REWARDS or the X-Force systems as they only interface through the (forced) concrete executions. Facilitated by X-Force, REWARDS is able to run legacy binaries and COTS binaries without any inputs. In our experiment, we run the new system on the 12 SPEC2000 INT binaries. They are a lot more complex than the Linux core-util programs used in the original paper [30]. To acquire the ground truth, we compile the programs with the option of generating debugging symbols as PDB files, and use DIA SDK to read the type information from the PDB files.

We evaluate the system in terms of both coverage and accuracy. Coverage means the percentage of variables in the program that have been executed by our system. Accuracy is the percentage of the covered variables whose types are correctly reverse engineered. From Fig. 8, the average coverage is around 84%. The coverage heavily relies on the code coverage of X-Force. Recall that these programs have non-trivial portion of unreachable code. The variables in those code regions cannot be reverse engineered by our system. From Fig.9, the average accuracy is about 90%. The majority of type inference

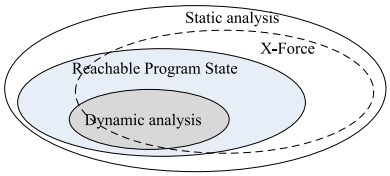


Figure 10: Essence of X-Force.

failures is caused by the fact that the variables are not related to any *type sink*.

We also compare with IDA and the original REWARDS. IDA has a static type inference algorithm that works in a similar fashion. When we run the original REWARDS, we have two configurations: (1) use the test input only (1 input per program) and (2) use both the test and the reference inputs (around 4 inputs per program). From Fig. 8 and Fig. 9, our system has much better accuracy than IDA (90% vs. 55% on average) and better coverage than the original REWARDS, i.e., 84% vs. 57% (test+reference) or 34% (test input only). The better accuracy than IDA is achieved by the more precise modeling of behavior difficult for static analysis, such as heap accesses and indirect calls and jumps.

6 Discussion and Future Work

X-Force is intended to be a *practical* solution for analyzing unknown (malicious) binaries without requiring any source code or inputs. Hence, X-Force trades soundness and completeness for practicality. It is unsound as it could explore infeasible paths. It is incomplete as it cannot afford exploring all paths. Figure 10 shows how X-Force compares with static and dynamic analysis: The "Reachable Program State" oval denotes all states that can be reached through possible program inputs – the ideal coverage for program analysis. Static analyses often make conservative approximations such that they yield *over-approximate* coverage. Dynamic analyses analyze a number of real executions and hence yield *under-approximate* results. X-Force explores a larger set of executions than dynamic analyses. Since X-Force makes unsound approximations, its results may be invalid (i.e., outside the ideal oval). Furthermore, it is incomplete as its results may not cover the ideal ones.

However, we argue that X-Force is still of importance in practice: (1) There are many security applications whose analysis results are not so sensitive to paths, such as the three studies in this paper. As such, path infeasibility may not affect the results much. However, having concrete states in path exploration is still critical in these applications such that an execution based approach like X-Force is suitable; (2) Only a very small percentage of predicates are switched (Section 5.1) in X-Force. Execution is allowed to proceed naturally in most predicates, respecting path feasibility. According to our observations, most of the predicates that got switched in linear search are those checking if the program has been provided the needed parameters, if files are properly opened, and if certain environmental configurations are correctly set-up; (3) In X-Force, taint analysis is used to identify predicates that are affected by inputs and only such predicates are eligible for switching.

Moreover, X-Force allows users to (1) *rapidly* explore the behaviors of any (unknown) binary as it simply executes the binary (without solving constraints); (2) handle binaries in a much *broader* spectrum (e.g., large, packed, or obfuscated binaries); (3) easily port or develop dynamic analysis on X-Force as the executions in X-Force are no different from regular concrete executions.

Future Work. We believe this paper is just an initial step in developing a unique type of program analysis different from the traditional static, dynamic, and symbolic analysis. We have a number of tasks in our future research agenda.

- While X-Force simply forces the branch outcomes of a few predicates without considering their feasibility, we suspect that there is a chance in practice the forced paths are indeed feasible in many cases. Note that the likelihood of infeasibility is not high if the forced predicates are not closely correlated. We plan to use a symbolic analysis engine that models the path conditions along the forced paths to observe how often they are infeasible.

- We develop 3 exploration algorithms in this paper. From the evaluation data on the SPECINT2000 programs, there are cases (e.g., perlbmk) that the current exploration algorithms cannot handle well. More effective algorithms, for example, based on modeling functions behaviors and caching previous exploration choices, will be developed.

- We currently handle multi-threaded programs by serializing their executions. In the future, we will explore forcing real concurrent executions. We envision this has to be integrated with flipping schedule decisions, which is a standard technique in exploring concurrent execution state. How to handle the enlarged state space and the potentially introduced infeasible thread schedules will be the new challenges.

- The current system is implemented as a tool on top of PIN. To build a tool that makes use of X-Force, for example REWARDS, the implementation of the additional tool is currently mixed with X-Force. They are compiled together to a single PIN-tool. We aim to make X-Force transparent to dynamic analysis developers by providing an PIN-like interface. Ideally, existing PIN-tools can be easily

ported to X-Force to benefit from the large number of executions provided by the X-Force engine.

- We also plan to port the core X-Force engine to other platforms such as mobile and HTML5 platforms.

7 Related Work

Researchers proposed to force branch outcomes for patching software failures in [51]. Hardware support was proposed to facilitate path forcing in [31]. Both require source code and concrete program inputs. Branch outcomes are forced to explore paths of binary programs in [48] to construct control flow graphs. The technique does not model any heap behavior. Moreover, it skips all library calls. Similar techniques are proposed to expose hidden behavior in Android apps [22, 45]. These techniques randomly determine each branch's outcome, posing the challenge of excessive infeasible paths. Forced execution was also proposed to identify kernel-level rootkits [46]. It completely disregards branch outcomes during execution and performs simple depth-first search. None of these techniques performs exception recovery and instead simply terminates executions when exceptions arise. Constraint solving was used in exploring execution paths to expose malware behavior in [33, 6]. They require concrete inputs to begin with and then mutate such inputs to explore different paths.

X-Force is related to static binary analysis [21, 3, 25, 42, 41], dynamic binary analysis [30, 39, 24] and symbolic binary analysis [10, 40]. We have discussed their differences from X-Force in Section 6, which are also supported by our empirical results in Section 5. X-Force is also related to failure oblivious computing [36] and on-the-fly exception recovery [34], which are used for failure tolerance and debugging and require source code.

8 Conclusion

We develop a novel binary analysis engine X-Force, which forces a binary to execute without any inputs or the needed environment. It systematically forces the branch outcomes at a small number of predicates to explore different paths. It can recover from exceptions by allocating memory on-demand and fixing correlated pointers accordingly. Our experiments on three security applications show that X-Force has similar precision as dynamic analysis but much better coverage due to the capability of exploring many paths with any inputs.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This work was motivated in part by the earlier research of Dr. Vinod Yegneswaran on brute-force malware execution and analysis. His influence and support is gratefully acknowledged. This research has been supported, in part, by DARPA under Contract 12011593 and by a gift from Cisco Systems. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

References

[1] Exposing the password secrets of internet explorer. `http://securityxploded.com/iepasswordsecrets.php`.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86–a platform for analyzing x86 executables. In *Proceedings of International Conference on Compiler Construction (CC)*, 2005.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of Intenational Conference on Compiler Construction (CC)*, 2004.

[4] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2010.

[5] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Blackhat USA'12.

[6] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. 2008.

[7] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS)*, 2007.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[9] M. I. Center. Apt1: Exposing one of chinas cyber espionage units. Technical report, 2013.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[11] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254, 2006.

[12] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium (Security)*, 2007.

[13] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[14] Z. Deng, X. Zhang, and D. Xu. Bistro: Binary component extraction and embedding for software security applications. In *18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[15] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[16] N. Falliere, L. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 2011.

[17] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, 2006.

[18] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[19] FireEye. Advanced targeted attacks: How to protect against the new generation of cyber attacks. In *White Paper*, 2013.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[21] Hex-Rays. Ida pro disassembler. `http://www.hex-rays.com/products/ida/index.shtml`.

[22] R. Johnson and A. Stavrou. Forced-path execution for android applications on x86 platforms. Technical report, Technical Report, Computer Science Department, George Mason University, 2013.

[23] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium (Security)*, 2009.

[24] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 29–44, 2010.

[25] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[26] F. Li. A detailed analysis of an advanced persistent threat malware. *SANS Institute*, 2011.

[27] J. Lim, T. Reps, and B. Liblit. Extracting file formats from executables. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, 2006.

[28] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[29] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[30] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[31] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, 2006.

[32] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected means of protocol inference. In *Proceedings of the 6th ACM SIGCOMM*

on Internet measurement (IMC), pages 313–326, 2006.

[33] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP)*, pages 231–245, 2007.

[34] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3), 2007.

[35] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Proceedings of the 10th international conference on Information Security (ISC)*. 2007.

[36] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004.

[37] N. Riva and F. Falcón. Dynamic binary instrumentation frameworks: I know you're there spying on me. In *RECON Conference*, 2012.

[38] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, 2005.

[39] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[40] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, 2008.

[41] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2000.

[42] H. Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA)*, 2000.

[43] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *2006 IEEE Symposium on Security and Privacy (SP)*, 2006.

[44] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of 14th European Symposium on Research in Computer Security (ESORICS)*, 2009.

[45] Z. Wang, R. Johnson, R. Murmuria, and A. Stavrou. Exposing security risks for commercial mobile devices. In *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security (MMM-ACNS)*, pages 3–21, 2012.

[46] J. Wilhelm and T.-c. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th international conference on Recent advances in intrusion detection (RAID)*, pages 219–235, 2007.

[47] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[48] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. Technical report, Technical Report CSE-2009-27, Department of Computer Science, UC Davis, 2009.

[49] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2e: Combing hardware virtualization and software emulation for transparent and extensible malware analysis. In *8th Annual International Conference on Virtual Execution Environments (VEE)*, 2012.

[50] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, 2007.

[51] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, 2006.

BYTEWEIGHT: Learning to Recognize Functions in Binary Code

Tiffany Bao
Carnegie Mellon University
tiffanybao@cmu.edu

Jonathan Burket
Carnegie Mellon University
jburket@cmu.edu

Maverick Woo
Carnegie Mellon University
pooh@cmu.edu

Rafael Turner
University of Chicago
turnersr@uchicago.edu

David Brumley
Carnegie Mellon University
dbrumley@cmu.edu

Abstract

Function identification is a fundamental challenge in reverse engineering and binary program analysis. For instance, binary rewriting and control flow integrity rely on accurate function detection and identification in binaries. Although many binary program analyses assume functions can be identified a priori, identifying functions in stripped binaries remains a challenge.

In this paper, we propose BYTEWEIGHT, a new automatic function identification algorithm. Our approach automatically learns key features for recognizing functions and can therefore easily be adapted to different platforms, new compilers, and new optimizations. We evaluated our tool against three well-known tools that feature function identification: IDA, BAP, and Dyninst. Our data set consists of 2,200 binaries created with three different compilers, with four different optimization levels, and across two different operating systems. In our experiments with 2,200 binaries, we found that BYTEWEIGHT missed 44,621 functions in comparison with the 266,672 functions missed by the industry-leading tool IDA. Furthermore, while IDA misidentified 459,247 functions, BYTEWEIGHT misidentified only 43,992 functions.

1 Introduction

Binary analysis is an essential security capability with extensive applications, including protecting binaries with control flow integrity (CFI) [1], extracting binary code sequences from malware [9], and hot patching vulnerabilities [25]. Research interest in binary analysis shows no sign of waning. In 2013 alone, several papers such as CFI for COTS [34] (referred to as COTS-CFI in this paper), the Rendezvous search engine for binaries [21], and the Phoenix decompiler [28] focus on developing new binary analysis techniques.

Function identification is a preliminary and necessary step in many binary analysis techniques and applications. For example, one property of CFI is to constrain inter-function control flow to valid paths. In order to reason about such paths, however, binary-only CFI infrastructures need to be able to identify functions accurately. In particular, COTS-CFI [34], CCFIR [33], MoCFI [12], Abadi et al. [1], and extensions like XFI [15] all depend on accurate function identification to be effective.

CFI is not the only consumer of binary-level function identification techniques. For example, Rendezvous [21] is a search engine that operates at the granularity of function binaries; incorrect function identification can therefore result in incomplete or even incorrect search results. Decompilers such as Phoenix [28], Boomerang [32], and Hex-Rays [18] recover high-level source code from binary code. Naturally, decompilation occurs on only those functions that have been identified in the input binary.

Given the foundational impact of accurate function identification in so many security applications, is this problem easy and can thus be regarded as "solved"? Interestingly, recent security research papers seem to have conflicting opinions on this issue. On one side, Kruegel et al. argued in 2004 that function start identification can be solved "very well" [23, §4.1] in regular binaries and even some obfuscated ones. On the other side, Perkins et al. described static function start identification as "a complex task in a stripped x86 executable" [25, §2.2.3] and therefore applied a dynamic approach in their ClearView system. A similar opinion is also shared by Zhang et al., who stated that "it is difficult to identify all function boundaries" [34, §3.2] and used a set of heuristics for this task.

So how good are the current tools at identifying functions from stripped, non-malicious binaries? To find out, we collected a dataset of 2,200 Linux and Windows binaries generated by GNU `gcc`, Intel `icc`, and Microsoft Visual Studio (`VS`) with multiple optimization levels. We then use our dataset to evaluate the most recent release of three popular off-the-shelf solutions for function identification: (i) IDA (v6.5 at submission), used in CodeSurfer/x86 [2], Choi et al.'s work on statically determining binary similarity [11], BinDiff [4], and BinNavi [5]; (ii) the CMU Binary Analysis Platform

(BAP, v0.7), used in the Phoenix decompiler [28] and the vulnerability analysis tool Mayhem [10]; and (iii) the `unstrip` utility in Dyninst (dated 2012-11-30), used in BinSlayer [7], Sharif et al.'s work on dynamic malware analysis [29], and Sidiroglou et al.'s work on software recovery navigation [30].

Our finding is that while IDA performs better than BAP and Dyninst on our dataset, its result can still be quite alarming—in our experiment, IDA returned 521,648 true positives (41.81%), 266,672 false negatives (21.38%), and 459,247 false positives (36.81%). While there is no doubt that such failures can have a negative impact on downstream security analyses, a real issue is in setting the right expectation on the subject within the security research community. If there is a publicly-available function identification solution where both its mechanism and limitations are well-understood by researchers, then researchers may be come up with creative strategies to cope with the limitations in their own projects. The goal of this paper is to explain our process of developing such a solution and to establish its quality through evaluating it against the aforementioned solutions.

We draw inspirations from how BAP and Dyninst perform function identification since their source code is available. Both solutions rely on fixed, manually-curated signatures. Dyninst, at the version we tested, uses the byte signature `0x55` (`push %ebp` in assembly) to recognize function starts in ELF x86 binaries [14]. BAP v0.7 uses a more complex signature, but it is also manually generated. Unfortunately, the process of manually generating such signatures do not scale well. For example, each new compiler release may introduce new idioms that require new signatures to capture. The myriad of different optimization settings, such as omit frame pointers, may also demand even more signatures. Clearly, we cannot expect to manually catch up.

One approach to recognizing functions is to automatically learn key features and patterns. For example, seminal work by Rosenblum et al. proposed binary function start identification as a supervised machine learning classification problem [27]. They model function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected idioms (patterns) appear in the CRF. Since standard inference methods for CRF on large, highly-connected graphs are expensive, Rosenblum et al. adopted feature selection and approximate inference to speed up their model. However, using hardware available in 2008, they needed 150 compute-days just for the feature selection phase on 1,171 binaries.

In this paper, we propose a new automated analysis for inferring functions and implemented it in our BYTEWEIGHT system. A key aspect of BYTEWEIGHT is the ability to learn signatures for new compilers and optimizations at least one order of magnitude faster than as reported by Rosenblum et al. [27], even after generously accounting for CPU speed increase since 2008. In particular, we avoid using CRFs and feature selection, and instead opt for a simpler model based on learning prefix trees. Our simpler model is scalable using current computing hardware: we finish training 2,064 binaries in under 587 compute-hours. BYTEWEIGHT also does not require compiler information of testing binaries, which makes the tool more powerful in practice. In the interest of open science, we also make our tools and datasets available to seed future improvements.

At a high level, we learn signatures for function starts using a weighted prefix tree, and recognize function starts by matching binary fragments with the signatures. Each node in the tree corresponds to either a byte or an instruction, with the path from the root node to any given node representing a possible sequence of bytes or instructions. The weights, which can be learned with a single linear pass over the data set, express the confidence that a sequence of bytes or instructions corresponds to a function start. After function start identification, we then use value set analysis (VSA) [2] with an incremental control flow recovery algorithm to find function bodies with instructions, and extract function boundaries.

To evaluate our techniques, we perform a large-scale experiment and provide empirical numbers on how well these tools work in practice. Based on 2,200 binaries across operating systems, compilers and optimization options, our results show that BYTEWEIGHT has a precision and recall of 97.30% and 97.44% respectively for function *start* identification. BYTEWEIGHT also has a precision and recall of 92.84% and 92.96% for function *boundary* identification. Our tool is adaptive for varying compilers and therefore more general than current pattern matching methods.

Contributions. This paper makes the following contributions:

- We enumerate the challenges we faced and implement a new function start identification algorithm based on prefix trees. Our approach is automatic and does not require a priori compiler information (see §4). Our approach models the function start identification problem in a novel way that makes it amenable to much faster learning algorithms.
- We evaluate our method on a large test suite across operating systems, compilers, and compiling optimizations. Our model achieves better accuracy than previously available tools.
- We make our test infrastructure, data set, implementation, and results public in an effort to promote open science (see §5).

```
1   #include <stdio.h>
2   #include <string.h>
3   #define MAX 10
4   void sum(char *a, char *b)
5   {
6       printf("%s + %s = %d\n",
7              a, b, atoi(a) + atoi(b));
8   }
9   void sub(char *a, char *b)
10  {
11      printf("%s - %s = %d\n",
12             a, b, atoi(a) - atoi(b));
13  }
14  void assign(char *a, char *b)
15  {
16      char pre_b[MAX];
17      strcpy(pre_b, b);
18      strcpy(b, a);
19      printf("b is changed from %s to %s\n",
20             pre_b, b);
21  }
22  int main(int argc, char **argv)
23  {
24      void (*funcs[3])(char *x, char *y);
25      int f;
26      char a[MAX], b[MAX];
27      funcs[0] = sum;
28      funcs[1] = sub;
29      funcs[2] = assign;
30      scanf("%d %s %s", &f, a, b);
31      (*funcs[f])(a, b);
32      return 0;
33  }
```

(a) Source code

```
1       00400660 <assign>:
2       mov    %rbx,-0x10(%rsp)
3       mov    %rbp,-0x8(%rsp)
4       sub    $0x28,%rsp
5       mov    %rdi,%rbp
6       lea    0xf(%rsp),%rdi
7       ...
8       004006b0 <sub>:
9       mov    %rbx,-0x18(%rsp)
10      mov    %rbp,-0x10(%rsp)
11      mov    %rsi,%rbx
12      mov    %r12,-0x8(%rsp)
13      xor    %eax,%eax
14      sub    $0x18,%rsp
15      ...
16      00400710 <sum>:
17      mov    %rbx,-0x18(%rsp)
18      mov    %rbp,-0x10(%rsp)
19      mov    %rsi,%rbx
20      mov    %r12,-0x8(%rsp)
21      xor    %eax,%eax
22      sub    $0x18,%rsp
23      ...
```

(b) Assembly compiled by `gcc -O3`

Figure 1: Example C Code. IDA fails to identify functions `sum`, `sub`, and `assign` in the compiled binary.

2 Running Example

We start with a simple example written in C, shown in Figure 1. In this program, three functions are stored as function pointers in the array `funcs`. When the program is run, input from the user dictates which function gets called, as well as the function arguments. We compiled this example code on Linux Debian 7.2 x86-64 using `gcc` with `-O3`, and stripped the binary using the command `strip`. We then used IDA to disassemble the binary and perform function identification. Many security tools use IDA in this way as a first step before performing additional analysis [9, 20, 24]. Unfortunately, for our example program IDA failed to identify the functions `sum`, `sub`, and `assign`.

IDA's failure to identify these three critical functions has significant implications for security analyses that rely on accurate function boundary identification. Recall that the CFI security policy dictates that runtime execution must follow a path of the static control flow graph (CFG). In this case, when the CFG is recovered by first identifying functions using IDA, any call to `sum`, `sub`, or `assign` would be incorrectly disallowed, breaking legitimate program behavior. Indeed, any indirect jump to an unidentified or mis-identified function will be blocked by CFI. The greater the number of functions missed, the more legitimate software functionality incorrectly lost. Secondly, suppose we are checking code for potential security-critical bugs. In our sample program, the function `assign` is vulnerable to a buffer overflow attack, but is not identified by IDA as a function. For tools like ClearView [25] that operate on binaries at the function level, missing functions can mean missing vulnerabilities.

In our analysis of 1,171 binaries, we observed that that IDA failed to identify 266,672 functions. BYTEWEIGHT improves on this number, missing only 44,621. BYTEWEIGHT also makes fewer mistakes, incorrectly identifying functions 43,992 times compared to 459,247 with IDA. While these results are not perfect, they demonstrate that our automated machine learning approach can outperform years of manual hand-tuning that has gone into IDA.

3 Problem Definition and Challenges

The goal of function identification is to faithfully determine the set of functions that exist in binary code. Determining what functions exist and which bytes belong to which functions is trivial if debug information is present.

For example, "unstripped" Linux binaries contain a symbol table that maps function names to locations in a binary, and Microsoft program database (PDB) information contains similar information for Windows binaries. We start with notation to make our problem definition precise and then formally define three function identification problems. We then describe several challenges to any approach or algorithm that addresses the function identification problems. In subsequent sections we provide our approach.

3.1 Notation and Definitions

A binary program is divided into a number of sections. Each section is given a type, such as code, data, read-only data, and so on. In this paper we only consider executable code, which we treat as a binary string.

Let B denote a binary string. For concreteness, think of this as a binary string from the `.text` section in a Linux executable. Let $B[i]$ denote the i^{th} byte of a binary string, and $B[i:i+j]$ refer to the list of contiguous bytes $B[i], B[i+1], \ldots, B[i+j-1]$. Thus, $B[i:i+j]$ is j-bytes long (with $j \geq 0$).

Each byte in an executable is associated with an *address*. The address of byte i is calculated with respect to a fixed section offset, i.e., if the section offset is ω, the address of byte i is $i+\omega$. For convenience, we omit the offset, and refer to i as the i^{th} address. Since the real address can always be calculated by adding the fixed offset, this can be done without loss of generality.

A function F_i in a binary B is a list of addresses corresponding to statements in either a function from the original compiled language or a function introduced directly by the compiler, denoted as

$$F = \{B[i], B[j], \ldots, B[k]\}$$

Note that function bytes need not be a set of contiguous addresses. We elaborate in §3.3 on real optimizations that result in high-level functions being compiled to a set of non-contiguous intervals of instructions.

Towards our goal of determining which bytes of a binary belong to which functions, we define the *set of functions* in a binary

$$\mathbf{FUNCS}(B) = \{F_1, F_2, \ldots, F_k\}.$$

Note that functions may share bytes, i.e., it may be that $F_1 \cap F_2 \neq \emptyset$. We give examples in §3.3 where this is the case.

We call the lowest address of a function F_i the *function start* address s_i, i.e., $s_i = \min(F_i)$. The *function end* address e_i is the maximum byte in a function body, i.e., $e_i = \max(F_i)$. We define the *function boundary* (s_i, e_i) as the function start and end addresses for F_i.

In order to evaluate function identification algorithms, we define ground truth in terms of oracles, which may have a number of implementations:

Function Oracle. $\mathbf{O}_{\text{func}}$ is an oracle that, given a binary B, returns a list of functions $\mathbf{FUNCS}(B)$ where each F_i is a set of bytes representing higher-level function i, as defined above.

Boundary Oracle. $\mathbf{O}_{\text{bound}}$ is an oracle that, given B, returns the set of function boundaries $\{(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\}$.

Start Oracle. $\mathbf{O}_{\text{start}}$ is an oracle that, given B, returns the set of function start addresses $\{s_1, s_2, \ldots, s_k\}$.

These oracles are successively less powerful. For example, implementing a boundary oracle $\mathbf{O}_{\text{bound}}$ from a function oracle $\mathbf{O}_{\text{func}}$ requires simply taking the minimum and maximum element of each F_i. Similarly, a start oracle $\mathbf{O}_{\text{start}}$ can be implemented from either $\mathbf{O}_{\text{func}}$ or $\mathbf{O}_{\text{bound}}$ by finding the minimum element of each F_i.

We do not restrict ourselves to a specific oracle implementation, as realizable oracles may vary across operating system and compiler. For example, the boundary oracle can be implemented by retaining debug information for Windows or Linux binaries. The function oracle can be implemented by instrumenting a compiler to output a list of instruction addresses included in each compiled function.

3.2 Problem Definition

With the above definitions, we are now ready to state our problem definitions. We start with the least powerful identification (function start) and build up to the most difficult one (entire function).

Definition 3.1. The *Function Start Identification* (FSI) problem is to output the complete list of function starts $\{s_1, s_2, \ldots, s_k\}$ given a binary B compiled from a source with k functions.

Suppose there is an algorithm $\mathscr{A}_{\mathsf{FSI}}(B)$ for the FSI problem which outputs $S = \{s_1, s_2, \ldots, s_k\}$. Then:

- The set of true positives, TP, is $S \cap \mathbf{O}_{\text{start}}(B)$.
- The set of false positives, FP, is $S - \mathbf{O}_{\text{start}}(B)$.
- The set of false negatives, FN, is $\mathbf{O}_{\text{start}}(B) - S$.

We also define precision and recall. Roughly speaking, precision reflects the number of times an identified function start is really a function start. A high precision means that most identified functions are indeed functions, whereas a low precision means that some sequences are incorrectly identified as functions. Recall is the measurement describing how many functions were identified within a binary. A high recall means an algorithm detected most functions, whereas a low recall means most functions were missed. Mathematically, they can be expressed as

$$\text{Precision} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FP}|}$$

and

$$\text{Recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}.$$

A more difficult problem is to identify both the start *and* end addresses for a function:

Definition 3.2. The *Function Boundary Identification* (FBI) problem is to identify the start and end bytes (s_i, e_i) for each function i in a binary, i.e., $S = \{(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\}$, given a binary B compiled from a source with k identified functions.

Suppose there is an algorithm $\mathscr{A}_{\text{FBI}}(B)$ for the FBI problem which outputs $S = \{(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\}$. We then define true positives, false positives, and false negatives similarly to above with the additional requirement that *both* the start and end addresses must match the output of the boundary oracle, i.e., for oracle output (s_{gt}, e_{gt}) and algorithm output $(s_{\mathscr{A}}, e_{\mathscr{A}})$, a positive match requires $s_{gt} = s_{\mathscr{A}}$ *and* $e_{gt} = e_{\mathscr{A}}$. A false negative occurs if either the start or end address is wrong. Precision and recall are defined analogously to the FSI problem.

Finally, we define the general function identification problem:

Definition 3.3. The *Function Identification* (FI) problem is to output a set $\{F_1, F_2, \ldots, F_k\}$ where each F_i is a list of bytes corresponding to high-level function i given a binary B with k identified functions.

We define true positives, false positives, false negatives, precision, and recall for the FI problem in the same ways as FSI and FBI but add the requirement that all bytes of a function must be matched between agorithm and oracle.

The above problem definitions form a natural hierarchy, where function start identification is the easiest and full function identification is the most difficult. For example, an algorithm $\mathscr{A}_{\text{FBI}}$ for function boundaries can solve the function start problem by returning the start element of each tuple. Similarly, an algorithm for the function identification problem needs only return the maximum and minimum element to solve the function boundary identification problem.

3.3 Challenges

Identifying functions in binary code is made difficult by optimizing compilers, which can manipulate functions in unexpected ways. In this section we highlight several challenges posed by the behavior of optimizing compilers.

Not every byte belongs to a function. Compilers may introduce extra instructions for alignment and padding between or within a function. This means that not every instruction or byte must belong to a function. For example, suppose we have symbol table information for a binary B. One naive algorithm is to first sort symbol-table entries by address, and then ascribe each byte between entry f_i and f_{i+1} as belonging to function f_i. This algorithm has appeared in several binary analysis platforms used in security research, such as versions of BAP [3] and BitBlaze [6]. This heuristic is flawed, however. For example, in Figure 2 lines 7–8 are not owned by any function.

```
1    <_func1>:
2    100000e20:   push   %rbp
3    100000e21:   mov    %rsp,%rbp
4    100000e24:   lea    0x69(%rip),%rdi
5    100000e2b:   pop    %rbp
6    100000e2c:   jmpq   100000e5e <_puts$stub>
7    100000e31:   nopl   0x0(%rax)
8    100000e38:   nopl   0x0(%rax,%rax,1)
9    <func2>:
```

Figure 2: Unreachable function example: source code and assembly.

Functions may be non-contiguous. Functions may have gaps. The gaps can be jump tables, data, or even instructions for completely different functions [26]. As noted by Harris and Miller [19], function sharing code can also lead to non-contiguous functions. Figure 3 shows code that starts out with the function `ConvertDefaultLocale`. Midway through the function at lines 17–21, however, the compiler decided to include a few lines of code for `FindNextFileW` as an optimization. Many binary analysis platforms, such as BAP [3] and BitBlaze [6], are not able to handle non-contiguous functions.

Functions may not be reachable. A function may be dead code and never called, but nonetheless appear in the binary. Recognizing such functions is still important in many security scenarios. For example, suppose two malware samples both contain a unique, identifying, yet uncalled function. Then the two malware samples are likely related even though the function is never called. One consequence of this is that techniques based solely on recursive disassembling from program start are not well-suited to solve the function identification problem. A recursive disassembler only disassembles bytes that occur along some control flow path, and thus by definition will miss functions that are not called.

Unreachability may occur for several reasons, including compiler optimizations. For example, Figure 4 shows a function for computing factorials called `fac`. When compiled by `gcc -O3`, the result of the call to `fac` is precomputed and inlined. Although the code of `fac` appears, it is never called in the binary code.

Security policies such as CFI and XFI must be aware of all low-level functions, not just those in the original code.

```
1   <ConvertDefaultLocale>
2     7c8383ff:   mov    %edi,%edi
3     7c838401:   push   %ebp
4     ...
5     7c83840c:   jz     7c848556
6     7c838412:   test   %eax, %eax
7     7c838414:   jz     7c83965c
8     7c83841a:   mov    $1024,%ecx
9     7c83841f:   cmp    %ecx,%eax
10    7c838421:   jz     7c83965c
11    7c838427:   test   $252,%ah
12    7c83842a:   jnz    7c838442
13    7c83842c:   mov    %eax,%edx
14    ...
15    7c838442:   pop    %ebp
16    7c838443:   ret    4
17    ; chunk of different function FindNextFileW
18    7c838446:   push   6
19    7c838448:   call   sub_7c80935e
20    7c83844d:
21    ; end of chunk
22    ...
23    7c83965c:   call   GetUserDefaultLCID
24    7c890661:   jmp    7c838442
25    ...
26    7c848556:   mov    $8,%eax
27    7c84855b:   jmp    7c838442
```

Figure 3: Lines 17–21 show code from `FindNextFileW` included in the middle of `ConvertDefaultLocale`.

Functions may have multiple entries. High-level languages use functions as an abstraction with a single entry. When compiled, however, functions may have multiple entries as a result of specialization. For example, the `icc` compiler with `-O1` specialized the `chown_failure_ok` function in GNU LIBC. As shown in Figure 5, a new function entry `chown_failure_ok.` (note the period) is added for use when invoking `chown_failure_ok` with `NULL`. The compiled binary has both symbol table entries. Unlike shared code for two functions that were originally separate, the compiler here has introduced shared code via multiple entries as an optimization.

Identifying both functions is necessary in many security scenarios, e.g., CFI needs to identify each function entry point for safety, and realize that both are possible targets. More generally, any binary rewriting for protection (e.g., memory safety, control safety, etc.) would need to reason about both entry points.

Functions may be removed. Functions can be removed by function inlining, especially small functions. Compilers perform function-inlining to reduce function call overhead and expose more optimization opportunities. For example, the function `utimens_symlink` is inlined into the function `copy_internal` when compiled by `gcc` with `-O2`. The source code and assembly code are shown in Figure 6. Note that function inlining does not have to be explicitly declared with `inline` annotation in source code. Many compilers inline functions by default unless explicitly disabled with options such as `-fno-deault-inline` [17]. This indicates that for those binary analysis techniques which need function information, even though source code is accessible, a robust function identification technique should still operate on the program binary. If using source code, function identification may be less precise due to functions that are inlined during compilation.

Each compilation is different. Binary code is not only heavily influenced by the compiler but also the compiler version and specific optimizations employed. For example, `icc` does not pre-compute the result of `fac` in Figure 4, but `gcc` does. Even different versions of a compiler may change code. For example, traditionally `gcc` (e.g., version 3) would only omit the use of the frame pointer register `%ebp` when given the `-fomit-frame-pointer` option. Recent versions of `gcc` (such as version 4.2), however, opportunistically omit the frame pointer when compiled with `-O1` and `-O2`. As a result several tools that identified functions by scanning for `push %ebp` break. For example, Dyninst, used for instrumentation in several security projects, relies on this heuristic to identify functions and breaks on recent versions of `gcc`.

4 BYTEWEIGHT

In this section, we detail the design and algorithms used by BYTEWEIGHT to solve the function identification problems. We first start with the FSI problem, and then move to the more general function identification problem.

We cast FSI as a machine learning classification problem where the goal is to label each byte of a binary as either a function start or not. We use machine learning to automatically generate literal patterns so that BYTEWEIGHT can handle new compilers and new optimizations without relying on manually generated patterns or heuristics. Our algorithm works with both byte sequences and disassembled instruction sequences.

Our overall system is shown in Figure 7. Like any classification problem, we have a training phase followed by a classification phase. During training, we first compile a reference corpus of source code to produce binaries where the start addresses are known. At a high level, our algorithm creates a weighted prefix tree of known function start byte or instruction sequences. We *weight* vertices in the prefix tree by computing the ratio of true positives to the sum of true and false positives for each sequence in the reference data set. We have designed and implemented two variations of BYTEWEIGHT: one working with raw bytes and one with normalized disassembled instructions. Both use the same overall algorithm and data structures. We show in our evaluation that the normalization approach provides higher precision and recall, and costs less time (experiment 5.2).

In the classification phase, we use the weighted prefix tree to determine whether a given sequence of bytes or

```
1       int fac(int x)
2       {
3         if (x == 1) return 1;
4         else return x * fac(x - 1);
5       }
6
7       void main(int argc, char **argv)
8       {
9         printf("%d", fac(10));
10      }
```

(a) Source code

```
1       080483f0 <fac>:
2       ...
3       08048410 <main>:
4       ...
5       movl   $0x375f00,0x4(%esp)
6       movl   $0x8048510,(%esp)
7       call   8048300  ;call printf without fac
8       xor    %eax,%eax
9       add    $0x8,%esp
10      pop    %ebp
11      ret
```

(b) Assembly compiled by `gcc -O2`

Figure 4: Unreachable code: source code and assembly.

```
1   extern bool
2   chown_failure_ok (struct cp_options const *x)
3   {
4     return ((errno == EPERM || errno == EINVAL)
5             && !x->chown_privileges);
6   }
```

(a) Source Code

```
1   <chown_failure_ok>:
2   804f544:        mov    0x4(%esp),%eax
3   <chown_failure_ok.>:
4   804f548:        push   %esi
5   804f549:        push   %esi
6   804f54a:        push   %esi
7   ...
```

(b) Assembly compiled by `icc -O1`

Figure 5: `chown_failure_ok` is specialized: source code and assembly.

instructions corresponds to a function start. We say that a sequence corresponds to a function start if the corresponding *terminal node* in the prefix tree has a weight value larger than the threshold t. In the case where the sequence exactly matches a path in the prefix tree, the terminal node is the final node in this path. If the sequence does not exactly match a path in the tree, the terminal node is the last matched node in the sequence.

Once we identify function starts, we infer the remaining bytes (and instructions) that belong to a function using a CFG recovery algorithm. The algorithm incrementally determines the CFG using a variant of VSA [2]. If an indirect jump depends on the value of a register, then we over-approximate a solution to the function identification problem by adding edges that correspond to locations approximated using VSA.

4.1 Learning Phase

The input to the learning phase is a corpus of training binaries $\mathbb{T}$, and a maximum sequence length $\ell > 0$. ℓ serves as a bound on the maximum tree height.

In BYTEWEIGHT, we first generate the oracle $\mathbf{O}_{\text{bound}}$ by compiling known source using a variety of optimization levels while retaining debug information. The debug information gives us the required (s_i, e_i) pair for each function i in the binary.

In this paper, we consider two possibilities: learning over raw bytes and learning over normalized instructions. We refer to both raw bytes and instructions as a sequence of elements. The sequence length ℓ determines how many raw sequential bytes or instructions we consider for training.

Step 1: Extract first ℓ elements for each function (Extraction). In the first step, we iterate over all (s_i, e_i) pairs and extract the first ℓ elements. If there are fewer than ℓ elements in the function, we extract the maximum number of elements. For raw bytes, this is $B[s : s+\ell]$ bytes, and for instructions, it is the first ℓ instructions disassembled linearly starting from $B[s]$.

Step 2: Generate a prefix tree (Tree Generation). In step 2, we generate a *prefix tree* from the extracted sequences to represent all possible function start sequences up to ℓ elements.

A prefix tree, also called a trie, is a data structure enabling efficient information retrieval. In the tree, each non-root node has an associated byte or instruction. The sequence for a node n is represented by the elements that appear on the path from the root to n. Note that the tree represents all strings up to ℓ elements, not just exactly ℓ elements.

Figure 8a shows an example tree on instructions, where node `callq 0x43a28` represents the instruction sequence:

```
push   %ebp            ;saved stack pointer
mov    %esp,%ebp       ;establish new frame
callq  0x43a28         ;call another function
```

If the sequence is over bytes, the prefix tree is calculated directly, although our experiments indicate that a prefix tree calculated over normalized instructions fairs

```
1   static inline int
2   utimens_symlink (char const *file,
3                    struct timespec const *timespec)
4   {
5     int err = lutimens (file, timespec);
6     if (err && errno == ENOSYS)
7       err = 0;
8     return err;
9   }
10
11  static bool
12  copy_internal (char const *src_name,
13                 char const *dst_name,
14                 ...)
15  {
16    ...
17    if ((dest_is_symlink
18         ?utimens_symlink (dst_name,
19                           timespec)
20         :utimens (dst_name, timespec))
21         != 0)
22    ...
23  }
```

(a) Source Code

```
1   <_copy_internal>:
2   100003170:      push   %rbp
3   100003171:      mov    %rsp,%rbp
4   100003174:      push   %r15
5   100003176:      push   %r14
6   ...
7   10000468c:      test   %r14b,%r14b
8   10000468f:      je     100005bfd
9   100004695:      lea    -0x738(%rbp),%rsi
10  10000469c:      mov    -0x750(%rbp),%rdi
11  1000046a3:      callq  10000d020 <_lutimens>
12  1000046a8:      test   %eax,%eax
13  1000046aa:      mov    %eax,%ebx
14  ...
```

(b) Assembly compiled by `gcc -O2`

Figure 6: Example of function being removed due to function inlining optimization.

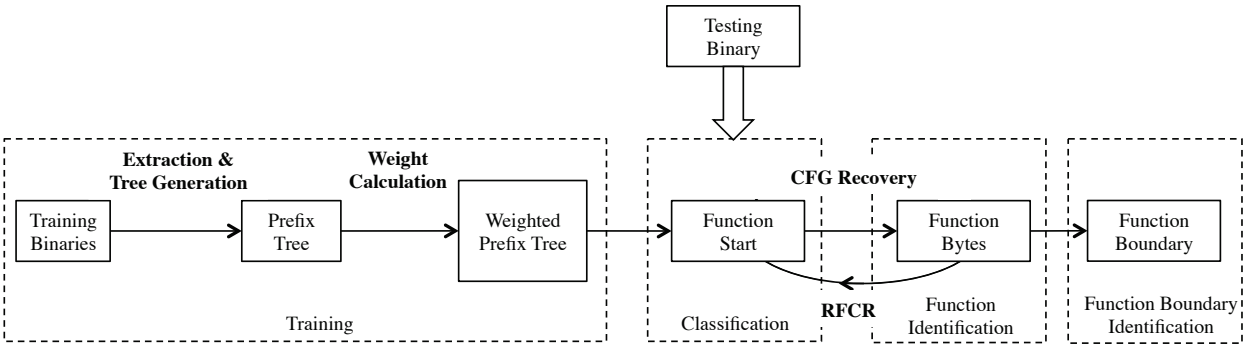


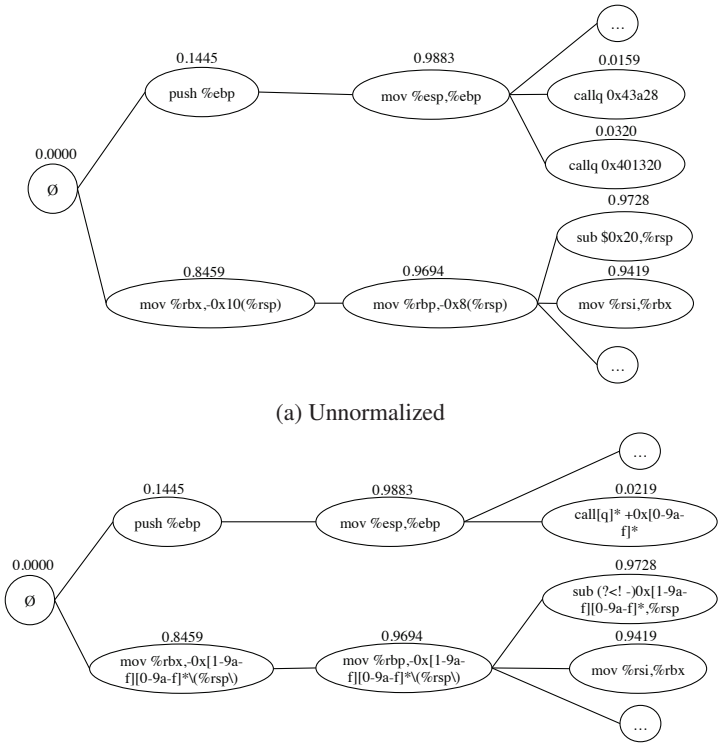
Figure 7: The BYTEWEIGHT function boundary inference approach.

better. We perform two types of normalization: immediate number normalization and call & jump instruction normalization. As shown in Table 1, normalization takes an instruction as input and generalizes it so that it can match against very similar, but not identical instructions. These two types of normalization help us improve recall at the cost of a little precision (Table 2). In our running example, only the function `assign` is recognized as a function start when matched against the unnormalized prefix tree (Figure 8a), while functions `assign`, `sub`, and `sum` can all be recognized when matched against the normalized prefix tree (Figure 8b).

Step 3: Calculate tree weights (Weight Calculation). The prefix tree represents possible function start sequences up to ℓ elements. For each node, we assign a weight that represents the likelihood that the sequence corresponding to the path from the root node to this node is a function start in the training set. For example, according to Figure 8, the weight of node `push %ebp` is 0.1445, which means that during training, 14.45% of all sequences with prefix of `push %ebp` were truly function starts, while 85.55% were not.

To calculate the weight, we first count the number of occurrences $\mathbb{T}_+$ in which each prefix in the tree matches a true function start with respect to the ground truth $\mathbf{O}_{\text{start}}$ for the entire training set $\mathbb{T}$.

Second, we lower the weight of a prefix if it occurs in a binary, but is not a function start. We do this by performing an exhaustive disassembly starting from every address that is *not* a function start [23]. We match each exhaustive disassembly sequence of ℓ elements against the tree. We call these *false matches*. The number of false matches $\mathbb{T}_-$ is the number of times a prefix represented



(a) Unnormalized

(b) Normalized

Figure 8: Example of unnormalized (a) and normalized (b) prefix tree. Weight is shown above its corresponding node.

in the tree is not a function start in the training set $\mathbb{T}$. The weight for each node n is then the ratio of true positives to overall matches

$$W_n = \frac{\mathbb{T}_+}{\mathbb{T}_+ + \mathbb{T}_-}. \quad (1)$$

Since the prefix tree can end up being quite large, it is beneficial to prune the tree of unnecessary nodes. For each node in the tree, we remove all its child nodes if the value of $\mathbb{T}_-$ for this node is 0. For any child node, the value of $\mathbb{T}_-$ is never negative and never larger than the value of $\mathbb{T}_-$ for the parent node. Hence, if $\mathbb{T}_-$ is 0 for a parent node, then the value must be 0 for all of the child nodes as well. The intuition here is that if a child node matches a sequence that is not a function start, then so must the parent node. Thus, if the parent node does not have any false matches, then neither can a child node. Based on Equation 1, if $\mathbb{T}_- = 0$ and $\mathbb{T}_+ > 0$, then the weight of the node is 1. Since the child nodes of such a node also have a $\mathbb{T}_-$ value of 0 and are not included in the tree if $\mathbb{T}_+ = 0$, they must also have a weight of 1. As discussed more in Section 4.2, child nodes with identical weights are redundant and can safely be removed without affecting classification.

This pruning optimization helps us greatly reduce the space needed by the tree. For example, pruning reduced the number of nodes in the prefix tree from 2,483 to 1,447 for our Windows x86 dataset. Moreover, pruning increases the speed of matching, since we can determine the weight of test sequences after traversing fewer nodes in the tree.

4.2 Classification Phase Using a Weighted Prefix Tree

The output of the learning phase is a weighted prefix tree (e.g., Figure 8). The input to the classification step is a binary B, the weighted prefix tree, and a weight threshold t.

To classify instructions, we perform exhaustive disassembly of the input binary B and match against the tree. Matching is done by tokenizing the disassembled stream, performing normalization as done during learning, and walking the tree. To classify bytes rather than instructions, we again start at every offset but instead match the raw bytes instead of normalized instructions.

The weight of a sequence is determined by last matching node (the *terminal* node) during the walk. For example, given the tree in Figure 8a, and our running example with sequences

```
mov    %rbx,-0x10(%rsp)
mov    %rbp,-0x8(%rsp)
sub    %0x28,%rsp
```

Type		Unnormalized Signature	Normalized Signature
Immediate	all	`mov $0xaa,%eax`	`mov \$-*0x[0-9a-f]+,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:-*0x[0-9a-f]+,%eax`
		`mov 0x80502c0,%eax`	`mov -*0x[0-9a-f]+,%eax`
	zero	`mov $0xaa,%eax`	`mov \$-*0x[1-9a-f][0-9a-f]*,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:0x0+,%eax`
		`mov 0x80502c0,%eax`	`mov -*0x[1-9a-f][0-9a-f]*,%eax`
	positive	`mov $0xaa,%eax`	`mov \$(?<! -)0x[1-9a-f][0-9a-f]*,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:-0x[0-9a-f]+|0x0+,%eax`
		`mov 0x80502c0,%eax`	`mov (?<! -)0x[1-9a-f][0-9a-f]*,%eax`
	negative	`mov $0xaa,%eax`	`mov \$(?<! -)0x[0-9a-f]+,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:(?<! -)0x[0-9a-f]+,%eax`
		`mov 0x80502c0,%eax`	`mov (?<! -)0x[0-9a-f]+,%eax`
		`movzwl -0x6c(%ebp),%eax`	`movzl -0x[1-9a-f][0-9a-f]*\(%ebp\),%eax`
	npz	`mov $0xaa,%eax`	`mov \$(?<! -)0x[1-9a-f][0-9a-f]*,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:0x0+,%eax`
		`mov 0x80502c0,%eax`	`mov (?<! -)0x[1-9a-f][0-9a-f]*,%eax`
		`movzwl -0x6c(%ebp),%eax`	`movzl -0x[1-9a-f][0-9a-f]*\(%ebp\),%eax`
Call & Jump		`call 0x804cf32`	`call[q]* +0x[0-9a-f]*`

For immediate normalization, we generalize immediate operands. There are five kinds of generalization: all, zero, positive, negative, and npz. For jump and call instruction normalization, we generalize callee and jump addresses.

Table 1: Normalizations in signature.

the matching node will be `mov%rbp,-0x8(%rsp)`, giving a weight of 0.9694. However, for another sequence

```
push   %ebp
and    $0x2,%esp
```

we would have weight 0.1445. We say the sequence is the beginning of a function if the output weight w is not less than the threshold t.

4.3 The Function Identification Problem

At a high level, we address the function identification problem by first determining the start addresses for functions, and then performing static analysis to recover the CFG of instructions that are reachable from the start. Direct control transfers (e.g., direct jumps and calls) are followed using recursive disassembly. Indirect control transfers, e.g., from indirect calls or jump tables, are enumerated using VSA [2]. The final CFG then represents all instructions (and corresponding bytes) that are owned by the function starting at the given address.

CFG recovery starts at a given address and recursively finds new nodes that are connected to found nodes. The process ends when no more vertices are added into graph. Starting at the addresses classified for FSI, CFG recovery recursively adds instructions that are reachable from these starts. A first-in-first-out vertex array is maintained during CFG recovery.

At the beginning, there is only one element – the start address in the array. In each round, we process the first element by exploring new reachable instructions. If the new instruction is not in the array, it will be appended to the end. Elements in the array are handled accordingly until all elements have been processed and no more instructions are added.

If the instruction being processed is a branch mnemonic, the reachable instruction is the branch reference. If it is a call mnemonic, the reachable instructions include both the call reference and the instruction directly following the call instruction. If it is an exit instruction, there will be no new instruction. For the rest of mnemonics, the new instruction is the next one by address. We handle indirect control transfer instruction by VSA: we infer a set that over-approximates the destination of the indirect jump and thus over-approximate the function identification problem.

Note that functions can exit by calling a no-return function such as `exit`. This means that some call instructions in fact never return. To detect these instances, we check the call reference to see if it represents a known no-return function such as `abort` or `exit`.

4.4 Recursive Function Call Resolution

Pattern matching can miss functions; for example, a function that is written directly in assembly may not obey calling conventions. To catch these kinds of missed functions, we continue to supplement the function start list during CFG recovery. If a call instruction has its callee in

the `.text` section, we consider the callee to be a function start. We then do CFG recovery again, starting at the new function start until there are no more functions added into the function start list. We will refer to this strategy as recursive function call resolution (RFCR). In §5.3, we discuss the effectiveness of this technique in function start identification.

4.5 Addressing Challenges

In this section, we describe how BYTEWEIGHT addresses the challenges raised in §3.3.

First, BYTEWEIGHT recovers functions that are unreachable via calls because it does not depend on calls to identify functions. In particular, BYTEWEIGHT recovers any function start that matches the learned weighted prefix tree as described above. Similarly, our approach will also learn functions that have multiple entries, provided a similar specialization occurs in the training set. This seems realistic in many scenarios since the number of compiler optimizations that create multiple entry functions are relatively few and can be enumerated during training.

BYTEWEIGHT also deals with overlapping byte or instruction sequences provided that there is a unique start address. Consider two functions that start at different addresses, but contain the same bytes. During CFG recovery, BYTEWEIGHT will discover that both functions use the same bytes, and attribute the bytes to both functions. BYTEWEIGHT can successfully avoid false identification for inlined functions when inlined function does not behave like an empirical function start (does not weighted over threshold in training).

Finally, note that BYTEWEIGHT does not need to attribute every byte or instruction to a function. In particular, only bytes (or instructions) that are reachable from the recovered function entries will be owned by a function in the final output.

5 Evaluation

In this section, we discuss our experiments and performance. BYTEWEIGHT is a cross-platform tool which can be run on both Linux and Windows. We used BAP [3] to construct CFGs. The rest of the implementation consists of 1988 lines of OCaml code and 222 lines of shell code. We set up BYTEWEIGHT on one desktop machine with a quad-core 3.5GHz i7-3770K CPU and 16GB RAM. Our experiments aimed to address three questions:

1. Does BYTEWEIGHT's pattern matching model perform better than known models for function start identification? (§5.2)
2. Does BYTEWEIGHT perform function start identification better than existing binary analysis tools? (§5.3)
3. Does BYTEWEIGHT perform function boundary identification better than existing binary analysis tools? (§5.4)

In this section, we first describe our data set and ground truth (the oracle), then describe the results of our experiments. We performed three experiments answering the above three questions. In each experiment, we compared BYTEWEIGHT against existing tools in terms of both accuracy and speed.

Because BYTEWEIGHT needs training, we divided the data into training and testing sets. We used standard 10-fold validation, dividing the element set into 10 sub-sets, applying 1 of the 10 on testing, and using the remaining 9 for training. The overall precision and recall represent the average of each test.

5.1 Data Set and Ground Truth

Our data set consisted of 2,200 different binaries compiled with four variables:

Operating System. Our evaluation used both Linux and Windows binaries.

Instruction Set Architecture (ISA). Our binaries consisted of both x86 and x86-64 binaries. One reason for varying the ISA is that the calling convention is different, e.g., parameters are passed by default on the stack in Linux on x86, but in registers on x86-64.

Compiler. We used GNU `gcc`, Intel `icc`, and Microsoft `VS`.

Optimization Level. We experimented with the four optimization levels from no optimization to full optimization.

On Linux, our data set consisted of 2,064 binaries in total. The data set contained programs from `coreutils`, `binutils`, and `findutils` compiled with both `gcc` 4.7.2 and `icc` 14.0.1. On Windows, we used `VS` 2010, `VS` 2012, and `VS` 2013 (depending on the requirements of the program) to compile 68 binaries for x86 and x86-64 each. These binaries came from popular open-source projects: putty, 7zip, vim, libsodium, libetpan, HID API, and pbc (a library for protocol buffers). Note that because Microsoft Symbol Server releases only public symbols which do not contain information of private functions, we were unable to use Microsoft Symbol Server for ground truth and include Windows system applications in our experiment.

We obtained ground truth for function boundaries from the symbol table and PDB file for Linux and Windows binaries, respectively. We used `objdump` to parse symbol tables, and `Dia2dump` [13] to parse PDB files. Additionally, we extracted "thunk" addresses from PDB files. While most tools do not take thunks into account, IDA considers thunks in Windows binaries to be special functions. To get a fair result, we filtered out thunks from IDA's output using the list of thunks extracted from PDB files.

5.2 Signature Matching Model

Our first experiment evaluated the signature matching model for function start identification. We compared BYTEWEIGHT and Rosenblum et al.'s implementation in terms of both accuracy and speed. In order to equally evaluate the signature matching models, recursive function call resolution was not used in this experiment.

The implementation of Rosenblum et al. is available as a matching tool with 12 hard-coded signatures for `gcc` and 41 hard-coded signatures for `icc`. Their learning code was not available, nor was their dataset. Although they evaluated `VS` in their paper, the version of the implementation that we had did not support `VS` and was limited to x86. Each signature has a weight, which is also hard-coded. After calculating the probability for each sequence match, it uses a threshold of 0.5 to filter out function starts. Taking a binary and a compiler name (`gcc` or `icc`), it generates a list of function start addresses. To adapt to their requirements, we divide Linux x86 binaries into two groups by compiler, where each group consists of 516 binaries. We did 10-fold cross validation for BYTEWEIGHT, and use the same threshold as Rosenblum et al.'s implementation.

We also evaluated another two varieties of our model: one without normalization, and one with a maximum tree height of 3, which is same as the model used by Rosenblum et al. and BYTEWEIGHT (3), respectively.

Table 2 shows precision, recall, and runtime for each compiler and each function start identification model. From the table we can see that Rosenblum et al.'s implementation had an accuracy below 70%, while both BYTEWEIGHT-series models achieved an accuracy of more than 85%. Note that BYTEWEIGHT with 10-length and normalized signatures (the last row in table) performed particularly well, with an accuracy of approximately 97%, a more than 35% improvement over Rosenblum et al.'s implementation.

Table 2 also details the accuracy and performance differences among BYTEWEIGHT with different configurations. Comparing against the full configuration model (BYTEWEIGHT), the model with a smaller maximum signature length (BYTEWEIGHT (3)) performs slightly faster (3% improvement), but sacrifices 7% in accuracy. The model without signature normalization (BYTEWEIGHT (no-norm)) has only 1% higher precision but 6.68% lower recall, and the testing time is ten times longer than that of the normalized model.

5.3 Function Start Identification

The second experiment evaluated our full function start identification against existing static analysis tools. We compared BYTEWEIGHT (no-RFCR)—a version without recursive function call resolution, BYTEWEIGHT, and the following tools:

IDA. We used IDA 6.5, build 140116 along with the default FLIRT signatures. All function identification options were enabled.

BAP. We used BAP 0.7, which provides a `get_function` utility that can be invoked directly.

Dyninst. Dyninst offers the tool *unstrip* [31] to identify functions in binaries without debug information.

Naive Method. This matched simple `0x55` (`push %ebp` or `push %rbp`) and `0xc3` (`ret` or `retq`) signatures only.

We divided our data set into four categories: ELF x86, ELF x86-64, PE x86, and PE x86-64. Unlike the previous experiment, binaries from various compilers but the same target were grouped together. Overall, we had 1032 ELF x86 and ELF x86-64 binaries, and 68 PE x86 and PE x86-64 binaries. We evaluated these categories separately, and again applied 10-fold validation. During testing, we used the same score threshold $t = 0.5$ as in the first experiment.

Note that not every tool in our experiment supports all binary targets. For example, Dyninst does not support ELF x86-64, PE x86, or PE x86-64 binaries. We use "-" to indicate when the target is not supported by the tool. Also, we omitted 3 failures in BYTEWEIGHT, and 10 failures in Dyninst during this experiment. Due to a bug in BAP, BYTEWEIGHT failed in 3 `icc` compiled ELF x86-64 binaries: *ranlib* with `-O3`, *ld_new* with `-O2`, and *ld_new* with `-O3`. Dyninst failed in 8 `icc` compiled ELF x86-64 binaries and 2 `gcc` compiled ELF x86-64 binaries. The results of our experiment are shown in Table 3.

As evident in Table 3, BYTEWEIGHT achieved a higher precision and recall than BYTEWEIGHT without recursive function call resolution. BYTEWEIGHT performed above 96% in Linux, while all other tools all performed below 90%. In Windows, we have comparable performance to IDA in terms of precision, but improved results in terms of recall.

Interestingly, we found that the naive method was not able to identify any functions in PE x86-64. This is mainly because `VS` does not use `push %rbp` to begin a function; instead, it uses move instructions.

5.4 Function Boundary Identification

The third experiment evaluated our function boundary identification against existing static analysis tools. As in the last experiment, we compared BYTEWEIGHT, BYTEWEIGHT (no-RFCR), IDA, BAP, and Dyninst, classified binaries by their target, and applied 10-fold validation on each of the classes. The results of our experiment are shown in Table 4.

Our tool performed the best in Linux, and was comparable to IDA in Windows. In particular, for Linux binaries, BYTEWEIGHT and BYTEWEIGHT (no-RFCR) have both precision and recall above 90%, while IDA is below 73%.

	GCC			ICC		
	Precision	Recall	Time(sec)	Precision	Recall	Time(sec)
Rosenblum et al.	0.4909	0.4312	1172.41	0.6080	0.6749	2178.14
BYTEWEIGHT (3)	0.9103	0.8711	1417.51	0.8948	0.8592	1905.34
BYTEWEIGHT (no-norm)	0.9877	0.9302	19994.18	0.9727	0.9132	20894.45
BYTEWEIGHT	0.9726	0.9599	1468.75	0.9725	0.9800	1927.90

Table 2: Precision/Recall of different pattern matching models for function start identification.

	ELF x86	ELF x86-64	PE x86	PE x86-64
Naive	0.4217/0.3089	0.2606/0.2506	0.6413/0.4999	0.0000/0.0000
Dyninst	0.8877/0.5159	–	–	–
BAP	0.8910/0.8003	–	0.3912/0.0795	–
IDA	0.7097/0.5834	0.7420/0.5550	0.9467/0.8780	0.9822/0.9334
BYTEWEIGHT (no-RFCR)	0.9836/0.9617	0.9911/0.9757	0.9675/0.9213	0.9774/0.9622
BYTEWEIGHT	0.9841/0.9794	0.9914/0.9847	0.9378/0.9537	0.9788/0.9798

Table 3: Precision/Recall for different function start identification tools.

For Windows binaries, IDA achieves better results than BYTEWEIGHT with x86-64 binaries, but is slightly worse for x86 binaries.

5.5 Performance

Training. We compare BYTEWEIGHT against Rosenblum et al.'s work in terms of time required for training. Since we do not have access to either their training code or their training data, we instead compare the results based on the performance reported in paper. There are two main steps in Rosenblum et al.'s work. First, they conduct feature selection to determine the most informative idioms – patterns that either immediately precede a function start, or immediately follow a function start. Second, they train parameters of these idioms using a logistic regression model. While they did not provide the time for parameter learning, they did describe that feature selection required 150 compute *days* for 1,171 binaries. Our tool, however, spent only 586.44 compute *hours* to train on 2,064 binaries, including overhead required to setup cross-validation.

Testing. We list the performance of BYTEWEIGHT, IDA, BAP, and Dyninst for testing. As described in section 4, BYTEWEIGHT has three steps in testing: function start identification by pattern matching, function boundary identification by CFG and VSA, and recursive function call resolution (RFCR). We report our time performance separately, as shown in Table 5.

IDA is clearly the fastest tool for PE files. For ELF binaries, it takes a similar amount of time to use IDA and BYTEWEIGHT to identify function starts, however our measured times for IDA also include the time required to run other automatic analyses. BAP and Dyninst have better performance on ELF x86 binaries, mainly because they match fewer patterns than BYTEWEIGHT and do not normalize instructions. This table also shows that function boundary identification and recursive function call resolution are expensive to compute. This is mainly because we use VSA to resolve indirect calls during CFG recovery, which costs more than typical CFG recovery by recursive disassembly. Thus while BYTEWEIGHT with RFCR enabled has improved recall, it is also considerably slower.

6 Discussion

Recall that our tool considers a sequence of bytes or instructions to be a function start if the weight of the corresponding terminal node in the learned prefix tree is greater than 0.5. The choice to use 0.5 as the threshold was largely dictated by Rosenblum et al., who also used 0.5 as a threshold in their implementation. While this appears to be a good choice for achieving high precision and recall in our system, it is not necessarily the optimal value. In the future, we plan to experiment with different thresholds to better understand how this affects the accuracy of BYTEWEIGHT.

While there are similarities betwen Rosenblum et al.'s approach and ours, there are also several key differences that are worth highlighting:

- Rosenblum et al. considered sequences of bytes or instructions immediately preceding functions, called prefix idioms, as well the entry idioms that start a function. Our present model does not include prefix idioms. Rosenblumet al.'s experiments show prefix

	ELF x86	ELF x86-64	PE x86	PE x86-64
Naive	0.4127/0.3013	0.2472/0.2429	0.5880/0.4701	0.0000/0.0000
Dyninst	0.8737/0.5071	–	–	–
BAP	0.6038/0.6300	–	0.1003/0.0219	–
IDA	0.7063/0.5653	0.7284/0.5346	0.9393/0.8710	0.9811/0.9324
BYTEWEIGHT (no-RFCR)	0.9285/0.9058	0.9317/0.9159	0.9503/0.9048	0.9287/0.9135
BYTEWEIGHT	0.9278/0.9229	0.9322/0.9252	0.9230/0.9391	0.9304/0.9313

Table 4: Precision/Recall for different function boundary identification tools.

	ELF x86	ELF x86-64	PE x86	PE x86-64
Dyninst	2566.90	–	–	–
BAP	1928.40	–	3849.27	–
IDA*	5157.85	5705.13	318.27	371.59
BYTEWEIGHT-Function Start	3296.98	5718.84	10269.19	11904.06
BYTEWEIGHT-Function Boundary	367018.53	412223.55	54482.30	87661.01
BYTEWEIGHT-RFCR	457997.09	593169.73	84602.56	97627.44

* For IDA, performance represents the total time needed to complete disassembly and auto-analysis.

Table 5: Performance for different function identification tools (in seconds).

idioms increase accuracy in their model. In the future, we plan to investigate whether adding prefix matching to our model can increase its accuracy as well.

- Rosenblum et al.'s idioms are limited to at most 4 instructions [27, p. 800] due to scalability issues with forward feature selection. With our prefix tree model, we can comfortably handle longer instruction sequences. At present, we settle on a length of 10. In the future, we plan to optimize the length to strike a balance between training speed and recognition accuracy.
- Rosenblum et al.'s CRF model considers both positive and negative features. For example, their algorithm is designed to avoid identifying two function starts where the second function begins within the first instruction of the first function (the so-called "overlapping disassembly"). Although we consider both positive and negative features as well, in constrast the above outcome is feasible with our algorithm.

While our technique is not compiler-specific, it is based on supervised learning. As such, obtaining representative training data is key to achieving good results with BYTEWEIGHT. Since compilers and optimizations do change over time, BYTEWEIGHT may need to be retrained in order to accurately identify functions in this new environment. Of course, the need for retraining is a common requirement for every system based on supervised learning. This is applicable to both BYTEWEIGHT and Rosenblum et al.'s work, and underscores the importance of having a computationally efficient training phase.

Despite our tool's success, there is still room for improvement. As shown in Section 5, over 80% of BYTEWEIGHT failures are due to the misclassification of the end instruction for a function, among which more than half are functions that do not return and functions that call such no-return functions. To mitigate this, we could backward propagate information about functions that do not return to the functions that call them. For example, if function `f` always calls function `g`, and `g` is identified as a no-return function, then `f` should also be considered a no-return function. We could also use other abstract domains along with the strided intervals of VSA to increase the precision of our indirect jump analysis [2], which can in turn help us identify more functions more accurately.

One other scenario where BYTEWEIGHT currently struggles is with Windows binaries compiled with hot patching enabled. With such binaries, functions will start with an extra `mov %edi,%edi` instruction, which is effectively a 2-byte `nop`. A training set that includes binaries with hot patching can reduce the accuracy of BYTEWEIGHT. Because the extra instruction `mov %edi,%edi` is treated as the function start in binaries with hot patching, any subsequent instructions are treated as false matches. Thus, any sequence of instructions that would normally constitute a function start but now follows a `mov %edi,%edi` is considered to be a false match. Consider a hypothetical dataset where all functions start with `push %ebp; mov %esp,%ebp`, but half of the binaries

are compiled with hot patching and thus start functions with an extra `mov %edi,%edi`. Half of the time, the sequence `push %ebp; mov %esp,%ebp` will be treated as a function start, but in the other half it will not be treated as such, thus leaving the sequence with a weight of 0.5 in our prefix tree. In order to deal with this compiler peculiarity, we would need give special consideration to `mov %edi,%edi`, treating both this instruction and the instruction following it as a function start for the sake of training.

Although training BYTEWEIGHT for function start identification is relatively fast, training for function *boundary* identification is still quite slow. Profiling reveals that most of the time is spent building CFGs, and in particular resolving indirect jumps using VSA. In future work, we plan to explore alternative approaches that avoid VSA altogether.

Finally, obfuscated or malicious binaries which intentionally obscure function start information are out of scope of this paper.

7 Related Work

In addition to the already discussed Rosenblum et al. [27], there are a variety of existing binary analysis platforms tackle the binary identification problem. BitBlaze [6] assumes debug information. If no debug information is present, it treats the entire section as one function. BitBlaze also provides an interface for incorporating Hex Rays function identification information.

Dyninst [19] also offers tools, such as *unstrip* [31], to identify functions in binaries without debug information. Within the Dyninst framework, potential functions in the `.text` section are identified using the hex pattern `0x55` representing `push %ebp`. First, Dyninst will start at the entry address and traverse inter- and intra-procedural control flow. The algorithm will scan the gaps between functions and check if `push %ebp` is present. This does not preform well across different optimizations and operating systems.

IDA using proprietary heuristics and FLIRT [16] technique attempts to help security researchers recover procedural abstractions. However, updating the signature database requires an amount of manual effort that does not scale. In addition, because FLIRT uses a pattern matching algorithm to search for signatures, small variations in libraries such as different compiler optimizations or the use of different compiler versions, prevent FLIRT from recognizing important functions in a disassembled program. The Binary Analysis Platform (BAP) also attempts to provide a reliable identification of functions using custom-written signatures [8].

Kruegel et al. perform exhaustive disassembly, then use unigram and bigram instruction models, along with patterns, to identify functions [23]. Jakstab uses two predefined patterns to identify functions for x86 code [22, §6.2].

8 Conclusion

In this paper, we introduce BYTEWEIGHT, a system for automatically learning to identify functions in stripped binaries. In our evaluation, we show on a test suite of 2,200 binaries that BYTEWEIGHT outperforms previous work across two operating systems, two compilers, and four different optimizations. In particular, BYTEWEIGHT misses only 44,621 functions in comparison with the 266,672 functions missed by the industry-leading tool IDA. Furthermore, while IDA misidentifies 459,247 functions, BYTEWEIGHT misidentifies only 43,992 functions. To seed future improvements to the function identification problem, we are making our tools and dataset available in support of open science.

Acknowledgments

This material is based upon work supported by DARPA under Contract No. HR00111220009. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity—principles, implementations, and applications. *ACM Transactions on Information and System Security 13*, 1 (2009), 1–40.

[2] BALAKRISHNAN, G. *WYSINWYX: What You See Is Not What You Execute*. PhD thesis, University of Wisconsin-Madison, 2007.

[3] BAP: Binary analysis platform. http://bap.ece.cmu.edu/.

[4] BinDiff. http://www.zynamics.com/bindiff.html.

[5] BinNavi. http://www.zynamics.com/binnavi.html.

[6] BitBlaze: Binary analysis for computer security. http://bitblaze.cs.berkeley.edu/.

[7] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM Program Protection and Reverse Engineering Workshop* (2013), ACM.

[8] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), Springer, pp. 463–469.

[9] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Network and Distributed System Security Symposium* (2010), The Internet Society.

[10] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 380–394.

[11] CHOI, S., PARK, H., LIM, H.-I., AND HAN, T. A static birthmark of binary executables based on API call structure. In *Proceeding of the 12th Asian Computing Science Conference* (2007), Springer, pp. 2–16.

[12] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., STEFAN, N., AND SADEGHI, A.-R. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012), The Internet Society.

[13] Dia2dump Sample. http://msdn.microsoft.com/en-us/library/b5ke49f5.aspx.

[14] Dyninst API. http://www.dyninst.org/.

[15] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *Proceedins of the 7th Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 75–88.

[16] IDA FLIRT Technology. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[17] GCC—Function Inline. http://gcc.gnu.org/onlinedocs/gcc/Inline.html.

[18] GUILFANOV, I. Decompilers and beyond. In *BlackHat USA* (2008).

[19] HARRIS, L. C., AND MILLER, B. P. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News 33*, 5 (2005), 63–68.

[20] HU, X., CHIUEH, T.-C., AND SHIN, K. G. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), ACM, pp. 611–620.

[21] KHOO, W. M., MYCROFT, A., AND ANDERSON, R. Rendezvous: A search engine for binary code. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories* (2013), IEEE, pp. 329–338.

[22] KINDER, J. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[23] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium* (2004), USENIX, pp. 255–270.

[24] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 601–615.

[25] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically patching errors in deployed software. In *Proceedings of the ACM 22nd Symposium on Operating Systems Principles* (2009), ACM, pp. 87–102.

[26] ROSENBLUM, N. The new Dyninst code parser: Binary code isn't as simple as it used to be, 2006.

[27] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence* (2008), AAAI, pp. 798–804.

[28] SCHWARTZ, E., LEE, J., WOO, M., AND BRUMLEY, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium* (2013), USENIX, pp. 353–368.

[29] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 16th Network and Distributed System Security Symposium* (2008), Internet Society.

[30] SIDIROGLOU, S., LAADAN, O., KEROMYTIS, A. D., AND NIEH, J. Using rescue points to navigate software recovery. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), IEEE, pp. 273–280.

[31] Unstrip. http://www.paradyn.org/html/tools/unstrip.html.

[32] VAN EMMERIK, M. J., AND WADDINGTON, T. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering* (2004), IEEE, pp. 27–36.

[33] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 559–573.

[34] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium* (2013), pp. 337–352.

Optimizing Seed Selection for Fuzzing

Alexandre Rebert[‡,$] alex@forallsecure.com
Sang Kil Cha[‡] sangkilc@cmu.edu
Thanassis Avgerinos[‡] thanassis@cmu.edu
Jonathan Foote[†] jmfoote@cert.org
David Warren[†] dwarren@cert.org
Gustavo Grieco[§] gg@cifasis-conicet.gov.ar
David Brumley[‡] dbrumey@cmu.edu

[‡] Carnegie Mellon University [$] ForAllSecure, Inc. [§] CIFASIS-CONICET
[†] Software Engineering Institute CERT

Abstract

Randomly mutating well-formed program inputs or simply *fuzzing*, is a highly effective and widely used strategy to find bugs in software. Other than showing fuzzers find bugs, there has been little systematic effort in understanding the science of how to fuzz properly. In this paper, we focus on how to mathematically formulate and reason about one critical aspect in fuzzing: how best to pick seed files to maximize the total number of bugs found during a fuzz campaign. We design and evaluate six different algorithms using over 650 CPU days on Amazon Elastic Compute Cloud (EC2) to provide ground truth data. Overall, we find 240 bugs in 8 applications and show that the choice of algorithm can greatly increase the number of bugs found. We also show that current seed selection strategies as found in Peach may fare no better than picking seeds at random. We make our data set and code publicly available.

1 Introduction

Software bugs are expensive. A single software flaw is enough to take down spacecrafts [2], make nuclear centrifuges spin out of control [17], or recall 100,000s of faulty cars resulting in billions of dollars in damages [5]. In 2012, the software security market was estimated at $19.2 billion [12], and recent forecasts predict a steady increase in the future despite a sequestering economy [19]. The need for finding and fixing bugs in software before they are exploited by attackers has led to the development of sophisticated automatic software testing tools.

Fuzzing is a popular and effective choice for finding bugs in applications. For example, fuzzing is used as part of the overall quality checking process employed by Adobe [28], Microsoft [14], and Google [27], as well as by security companies and consultants to find bugs and vulnerabilities in COTS systems.

One reason fuzzing is attractive is because it is relatively straightforward and fast to get working. Given a target application P, and a set of seed input files S, the programmer needs to:

Step 1. Discover the command line arguments to P so that it reads from a file. Popular examples include `-f`, `-file`, and using `stdin`. This step can be manual, or automated in many cases using simple heuristics such as trying likely argument combinations.

Step 2. Determine the relevant file types for an application automatically. For example, we are unlikely to find many bugs fuzzing a PDF viewer with a GIF image. Currently this step is performed manually, and like the above step the manual process does not scale to large program bases.

Step 3. Determine a subset of seeds $S' \subseteq S$ to fuzz the program. For example, an analyst may consider the possible set of seeds S as every PDF available from a search engine. Clearly fuzzing on each seed is computationally prohibitive, thus a *seed selection strategy* is necessary. Two typical choices are choosing the set of seed files ad-hoc, e.g., those immediately handy, and by finding the minimal set of seeds necessary to achieve code coverage.

Step 4. Fuzz the program and reap risk-reducing, or profitable, bugs.

Throughout this paper we assume maximizing the number of unique bugs found is the main goal. We make no specific assumptions about the type of fuzzer, e.g., we do not assume nor care whether black-box, white-box, mutational, or any other type of fuzzing is used. For our

experiments, we use BFF, a typical fuzzer used in practice, though the general approach should apply to any fuzzer using seeds. Our techniques also make no specific assumptions about the fuzz scheduling algorithm, thus are agnostic to the overall fuzzing infrastructure. To evaluate seed selection strategies, we use popular scheduling algorithms such as round-robin, as well as the best possible (optimal) scheduling.

We motivate our research with the problem setting of creating a hypothetical fuzzing testbed for our system, called COVERSET. COVERSET periodically monitors the internet, downloads programs, and fuzzes them. The goal of COVERSET is to maximize the number of bugs found within a limited time period or budget. Since budgets are forever constrained, we wish to make intelligent design decisions that employ the optimal algorithms wherever possible. How shall we go about building such a system?

Realizing such an intelligent fuzzing system highlights several deep questions:

Q1. Given millions, billions, or even trillions of PDF files, which should you use when fuzzing a PDF viewer? More generally, what algorithms produce the best result for seed selection of $S' \subseteq S$ in step 3?

Q2. How do you measure the quality of a seed selection technique *independently* of the fuzzing scheduling algorithm? For example, if we ran algorithm A on seed set S_1 and S_2, and S_1 maximized bugs, we would still be left with the possibility that with a more intelligent scheduling algorithm A' would do better with S_2 rather than S_1. Can we develop a theory to justify when one seed set is better than another with the best possible fuzzing strategy, instead of specific examples?

Q3. Can we converge on a "good" seed set for fuzzing campaigns on programs for a particular file type? Specifically, if S' performs well on program P_1, how does it work on other similar applications $P_2, P_3, \ldots$? If there is one seed set that works well across all programs, then we would only need to precompute it once and forever use it to fuzz any application. Such a strategy would save immense time and effort in practice. If not, we will need to recompute the best seed set for each new program.

Our main contribution are techniques for answering the above questions. To the best of our knowledge, many of the above problems have not been formalized or studied systematically. In particular:

- We formalize, implement, and test a number of existing and novel algorithms for seed selection.
- We formalize the notion of ex post facto optimality seed selection and give the first strategy that provides an optimal algorithm even if the bugs found by different seeds are correlated.
- We develop evidence-driven techniques for identifying the quality of a seed selection strategy with respect to an optimal solution.
- We perform extensive fuzzing experiments using over 650 CPU days on Amazon EC2 to get ground truth on representative applications. Overall, we find **240** unique bugs in 8 widely-used applications, all of which are on the attack surface (they are often used to process untrusted input, e.g., images, network files, etc.), most of which are security-critical.

While our techniques are general and can be used on any data set (and are the main contribution of this work), our particular result numbers (as any in this line of research) are data dependent. In particular, our initial set of seed files, programs under test, and time spent testing are all important factors. We have addressed these issues in several ways. First, we have picked several applications in each file type category that are typical of fuzzing campaigns. This mitigates incorrect conclusions from a non-representative data set or a particularly bad program. Second, we have performed experiments with reasonably long running times (12 hour campaigns per file), accumulating over 650 CPU days of Amazon EC2 time. Third, we are making our data set and code available, so that: 1) others need not spend time and money on fuzzing to replicate our data set, 2) others can further analyze the statistics to dig out additional meaning (e.g., perform their own hypothesis testing), and 3) we help lower the barrier for further improvements to the science of vulnerability testing and fuzzing. For details, please visit: http://security.ece.cmu.edu/coverset/.

2 Q1: Seed Selection

How shall we select seed files to use for the fuzzer? For concreteness, we downloaded a set of seed files S consisting of $4,912,142$ distinct files and 274 file types from Bing. The overall database of seed files is approximately 6TB. Fuzzing each program for a sufficient amount of time to be effective across all seed files is computationally expensive. Further, sets of seed files are often duplicative in the behavior elicited during fuzzing, e.g., s_1 may produce the same bugs as s_2, thus fuzzing both s_1 and s_2 is wasteful. Which subset of seed files $S' \subseteq S$ shall we use for fuzzing?

Several papers [1, 11], presentations from well-respected computer security professionals [8, 22, 26], as well as tools such as Peach [9], suggest using executable code coverage as a seed selection strategy. The intuition is that many seed files likely execute the same code blocks,

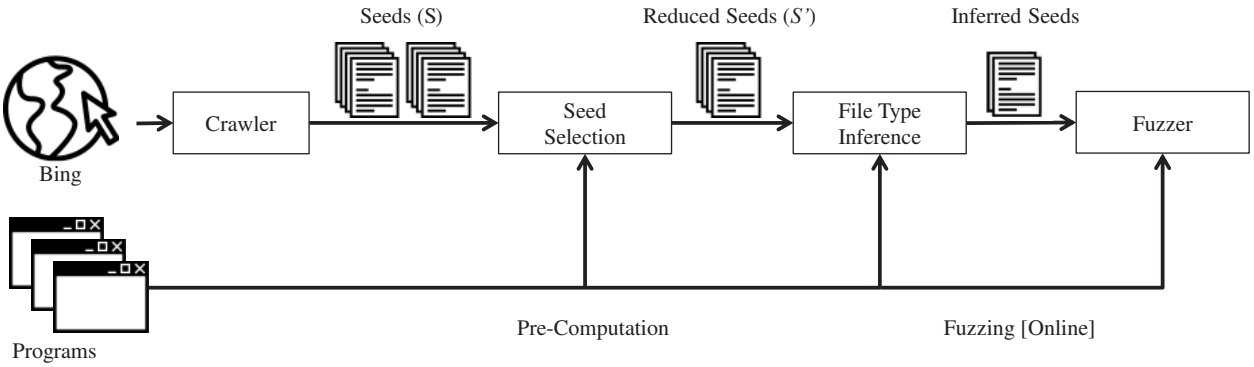


Figure 1: The COVERSET pipeline.

and such seeds are likely to produce the same bugs. For example, Miller reports a 1% increase in code coverage increases the percentage of bugs found by .92% [22]. This intuition can be formalized as an instance of the set cover problem [1, 11]. Does set cover work? Is the minimal set cover better than other set covers? Should we weight the set cover, e.g., by how long it takes to fuzz a particular seed? Previous work has shown a correlation between coverage and bugs found, but has not performed *comparative* studies among a number of approaches, nor studied how to measure optimality (§ 3).

Recall that in the *set cover problem* (SCP) [6] we are given a set X and a finite list of subsets $\mathbb{F} = \{S_1, S_2, \ldots, S_n\}$ such that every element of X belongs to at least one subset of $\mathbb{F}$:

$$X = \bigcup_{S \in \mathbb{F}} S$$

We say that a set $\mathbb{C} \subseteq \mathbb{F}$ is a set cover of X when:

$$X = \bigcup_{S \in \mathbb{C}} S$$

The seed selection strategy is formalized as:

Step 1. The user computes the coverage for each of the n individual seed files. The output is the set of code blocks [1] executed per seed. For example, suppose a user is given $n = 6$ seeds such that each seed executes the following code blocks:

$$S_1 = \{1,2,3,4,5,6\} \quad S_2 = \{5,6,8,9\}$$
$$S_3 = \{1,4,7,10\} \quad S_4 = \{2,5,7,8,11\}$$
$$S_5 = \{3,6,9,12\} \quad S_6 = \{10,11\}$$

Step 2. The user computes the cummulative coverage $X = \bigcup S_i$, e.g., $X = \{1,2,\ldots,12\}$ for the above.

[1] We assume code blocks, though any granularity of unit such as instruction, function, etc. also work.

Step 3. The user computes a set cover to output a subset $\mathbb{C}$ of seeds to use in a subsequent fuzzing campaign. For example, $\mathbb{C}_1 = \{S_1, S_4, S_3, S_5\}$ is one set cover, as is $\mathbb{C}_2 = \{S_3, S_4, S_5\}$, with $\mathbb{C}_2$ being optimal in the unweighted case.

The goal of the *minimal set cover problem* (MSCP) is to minimize the number of subsets in the set cover $\mathbb{C} \subseteq \mathbb{F}$. We call such a set $\mathbb{C}$ a *minset*. Note that a minset need not be unique, i.e., there may be many possible subsets of equal minimal cardinality. Each minset represents the fewest seed files needed to elicit the maximal set of instructions with respect to S, thus represents the maximum data seed reduction size.

In addition to coverage, we may also consider other attributes, such as speed of execution, file size, etc. A generalization of the set cover is to include a weight $w(S)$ for each $S \in \mathbb{F}$. The total cost of a set cover $\mathbb{C}$ is:

$$Cost\,(\mathbb{C}) = \sum_{S \in \mathbb{C}} w(S)$$

The goal of the *weighted* minimal set cover problem (WMSCP) is to find the minimal cost cover set, i.e., $\underset{\mathbb{C}}{argmin}\ Cost\,(\mathbb{C})$.

Both the MSCP and WMSCP can be augmented to take an optional argument k (forming k-SCP and k-WSCP respectively) specifying the maximum size of the returned solution. For example, if $k = 2$ then the number of subsets is restricted to at most 2 ($|\mathbb{C}| \leq 2$), and the goal is to *maximize* the number of covered elements. Note the returned set may not be a complete set cover.

Both MSCP and WMSCP are well-known NP-hard problems. Recall that a common approach to dealing with NP-hard problems in practice is to use an approximation algorithm. An approximation algorithm is a polynomial-time algorithm for approximating an optimal solution. Such an algorithm has an *approximation ratio* $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of

an optimal solution. The minimal set cover and weighted set cover problems both have a *greedy* polynomial-time $\ln|X|+1$-approximation algorithm [4, 16], which is a threshold below which set cover cannot be approximated efficiently assuming NP does not have slightly super-polynomial time algorithms, i.e., the greedy algorithm is essentially the best algorithm possible in terms of the approximation ratio it guarantees [10]. Since $\ln|X|$ grows relatively slowly, we expect the greedy strategy to be relatively close to optimal.

The optimal greedy polynomial-time approximation algorithm[2] for WSCP is:

GREEDY-WEIGHTED-SET-COVER$(X,\mathbb{F})$

```
1  U = X
2  ℂ = ∅
3  while U ≠ ∅
4      S = argmax_{S∈𝔽} |S∩U| / w(S)
5      ℂ = ℂ ∪ S
6      U = U \ S
7  return ℂ
```

Note that the unweighted minset can be solved using the same algorithm by setting $\forall S : w(S) = 1$.

2.1 Seed Selection Algorithms

In this section we consider: the set cover algorithm from Peach [9], a minimal set cover [1], a minimal set cover weighted by execution time, a minimal set cover weighted by size, and a hotset algorithm. The first two algorithms have previously been proposed in literature; the remaining are additional design points we propose and evaluate here. We put these algorithms to the test in our evaluation section to determine the one that yields the best results (see § 6).

All algorithms take the same set of parameters: given $|\mathbb{F}|$ seed files, the goal is to calculate a data reduction to k files where $k \ll |\mathbb{F}|$. We assume we are given t seconds to perform the data reduction, after which the selected k files will be used in a fuzzing campaign (typically of much greater length than t). We break ties between two seed files by randomly choosing one.

PEACH SET. Peach 3.1.53 [9] has a class called MinSet that calculates a cover set $\mathbb{C}$ as follows: [3]

PEACH-MINSET$(P,\mathbb{F})$

```
1  ℂ = ∅
2  i = 1
3  for S in 𝔽
4      cov[i] = MeasureCoverage( S )
5      i = i + 1
6  sort( cov ) // sort seeds by coverage
7  for i = 1 to |𝔽|
8      if cov[i] \ ℂ ≠ ∅
9          ℂ = ℂ ∪ cov[i]
10 return ℂ
```

Despite having the name MinSet, the above routine does not calculate the minimal set cover nor a proven competitive approximation thereof.

RANDOM SET. Pick k seeds at random. This approach serves as a baseline for other algorithms to beat. Since the algorithm is randomized, RANDOM SET can have high variance in terms of seed quality and performance. To measure the effectiveness of RANDOM SET, unless specified otherwise, we take the median out of a large number of runs (100 in our experiments).

HOT SET. Fuzz each seed for t seconds and return the top k seeds by number of unique bugs found. The rationale behind HOT SET is similar to multi-armed bandit algorithms—a buggy program is more likely to have more bugs. In our experiments, we fuzz each seeds for 5 minutes ($t = 300$) to compute the HOT SET.

UNWEIGHTED MINSET. Use an unweighted k-minset. This corresponds to standard coverage-based approaches [1, 23], and serves as a baseline for measuring their effectiveness. To compute UNWEIGHTED MINSET when k is greater than the minimum required to get full coverage, the minset is padded with files sorted based on the quality metric (coverage). We follow the same approach for TIME MINSET and SIZE MINSET.

TIME MINSET. Return a k-execution time weighted minset. This algorithm corresponds to Woo et al.'s observation that weighting by time in a multi-armed bandit fuzzing algorithm tends to perform better than the unweighted version [29]. The intuition is that seeds that are fast to execute ultimately lead to far more fuzz runs during a campaign, and thus potentially more bugs.

SIZE MINSET. Return a k-size weighted minset. Weighting by file size may change the ultimate minset, e.g., many smaller files that cover a few code blocks may be preferable to one very large file that covers many code blocks—both in terms of time to execute and bits to flip.

[2] Other algorithms exist to compute the weighted minset (see [6, 35-3.3]).

[3] This is a high-level abstraction of the `Delta` and `RunCoverage` methods. We checked the Peach implementation after the paper submission, and noticed that the sorting was removed (At Line 4 of the algorithm) in their MinSet implementation since Peach 3.1.95.

For example, SIZE MINSET will always select a 1KB seed over a 100MB seed, all other things being equal.

2.2 Specific Research Questions

Previous wisdom has suggested using UNWEIGHTED MINSET as the algorithm of choice for computing minsets [1, 23]. Is this justified? Further, computing the minset requires measuring code coverage. This computation requires time, time that could be spent fuzzing as in the HOT SET algorithm. Are coverage-based minsets beneficial and when?

More precisely, we formulate the following hypothesis:

Hypothesis 1 (MINSET > RANDOM.) *Given the same size parameter k,* MINSET *algorithms find more bugs than* RANDOM SET.

Hypothesis 1 is testing whether the heuristics applied by the algorithms presented above (§ 2.1) are useful. If they are as useful as choosing a random set, then the entire idea of using any of these MINSET algorithms is fundamentally flawed.

Hypothesis 2 (MINSET Benefits > Cost.) *Computing the* MINSET *for a given application and set of seed files and then starting fuzzing finds more bugs than just fuzzing.*

Hypothesis 2 tests whether the benefits of the minset outweigh the cost. Instead of spending time computing code coverage of seed files, should we instead spend it fuzzing? If yes, then the idea of reducing the files for *every* fuzzed application is flawed. It would also imply that precomputing minsets is necessary for the minsets to be useful. This observation leads to our next hypothesis.

Hypothesis 3 (MINSET Transferability.) *Given applications A and B that accept the same filetype F,* MINSET_F^A *finds the same or more bugs in application B as* MINSET_F^B.

Hypothesis 3 tests the transferability of seeds across applications that accept the same file type. For example, is the MINSET for PDF viewer A effective on PDF viewer B? If yes, we only need to compute a minset once per file type, thus saving resources (even if Hypothesis 2 is false).

Hypothesis 4 (MINSET Data Reduction.) *Given a target application A, a set of seed files F, and a* MINSET_F^A*, fuzzing with* MINSET_F^A *finds more bugs than fuzzing with the entire set F.*

Hypothesis 4 tests the main premise of using a reduced data set. Our MINSET contains fewer bugs than the full set in total. Under what conditions is the reduction beneficial?

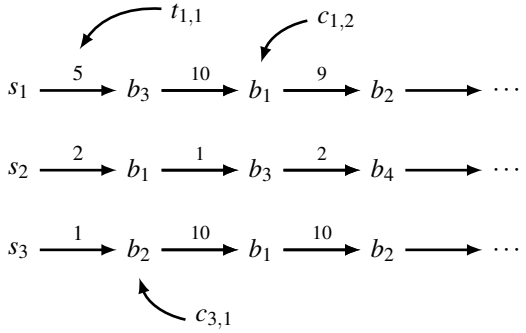


Figure 2: An example of output from fuzzing 3 seeds. Bugs may be found across seeds (e.g., b_1 is found by all seeds. A single seed may produce the same bug multiple times, e.g., with s_3. We also show the corresponding ILP variables t (interarrival times) and c (crash ids).

3 Q2: Measuring Selection Quality

There are a variety of seed selection strategies, e.g., to use minset or pick k seeds at random. How can we argue a particular seed selection strategy performs well?

One strawman answer is to run seed selection algorithm A to pick subset S_A, algorithm B to pick subset S_B. We then fuzz S_A and S_B for an equal amount of time and declare the fuzz campaign with the most bugs the winner. The fuzz campaign will incrementally fuzz each seed in each set according to its own scheduling algorithm. While such an approach may find the best seed selection for a *particular* fuzzing strategy, it provides no evidence that a particular subset is inherently better than another in the limit.

The main intuition in our approach is to measure the *optimal case* for bugs found with a particular subset of seeds. The best case provides an upper bound on any scheduling algorithm instead of on a particular scheduling algorithm. Note the lower bound on the number of bugs found for a subset is trivially zero, thus all we need is an upper bound.

To calculate the optimal case, we fuzz each seed in s_i for t seconds, recording as we fuzz the arrival rate of bugs. Given n seeds, the total amount of time fuzzing is $n * t$. For example, given 3 seeds we may have a bug b_i arrival time given by Figure 2.

Post-fuzzing, we then calculate the ex post facto optimal search strategy to maximize the number of bugs found. It may seem strange at first to calculate the optimal seed selection strategy after all seeds have been fuzzed at first blush. However, by doing so we can measure the quality of the seed selection strategy with respect to the optimal, thus give the desired upper bound. For example, if the seed selection strategy picks s_1 and s_2, we can calculate the maximum number of bugs that could be

found by any scheduler, and similarly for the set s_2, s_3 or any other set. Note we are calculating the upper bound to scientifically justify a particular strategy. For example, our experiments suggest to use UNWEIGHTED MINSET for seed selection. During a practical fuzzing campaign, one would not recompute the upper bound for the new dataset; instead, she would use the seed selection strategy that was shown to empirically perform best in previous tests.

3.1 Formalization

Let $Fuzz$ be a single-threaded fuzzer that takes in a set of seeds $\mathbb{C} \subseteq \mathbb{F}$ and a time threshold t_{thres} and outputs a sequence of unique bugs b_i along with the seed files that triggered them S_i and timestamps t_i:

$$Fuzz(\mathbb{C}, t_{\text{thres}}) = \{(b_1, S_1, t_1), \ldots, (b_n, S_n, t_n)\}$$

Given that we know the ground truth, i.e., we know the value of $Fuzz$ when applied on every singleton in $\mathbb{F}$: $Fuzz(\{S_i\}, t_{\text{thres}}) = \{(b_1, S_i, t_1), \ldots, (b_k, S_i, t_k)\}$, we can model the computation of the optimal scheduling/seed selection across all seed files in $\mathbb{F}$. Note that the ground truth is necessary, since any optimal solution can be only computed in retrospect (if we know how each seed would perform). We measure optimality of a scheduling/seed selection by computing the maximum number of unique bugs found.

The *optimal budgeted ex post facto scheduling problem* is given the ground truth for a set of seeds $Fuzz(\{S_i\}, t_{\text{thres}}) = \{(b_1, S_i, t_1), \ldots, (b_k, S_i, t_k)\}$ and a time threshold t_{thres}, automatically compute the interleaving of fuzzed seeds (time slice spent analyzing each one) to maximize the number of bugs found. The number of bugs found for a given minset gives an upper bound on the performance of the set and can be used as a quality indicator. Note that the same bug may be found by different seeds and may take different amounts of time to find.

Finding an optimal schedule for a given ground truth is currently an open problem. Woo et al. come the closest, but their algorithm assumes each seed produces independent bugs [29]. We observe finding an optimal scheduling algorithm is inherently an integer programming problem. We formulate finding the exact optimal seed scheduling as an Integer Linear Programming (ILP) problem. While computing the optimal schedule is NP-hard, ILP formulations tend to work well in practice.

First, we create an indicator variable for unique bugs found during fuzzing.

$$b_x = \begin{cases} 1 & \text{The schedule includes finding unique bug } x \\ 0 & \text{Otherwise} \end{cases}$$

The goal of the optimal schedule is to maximize the number of bugs. However, we do not see bugs, we see individual crashes arriving during fuzzing. We create an indicator variable $c_{i,j}$ that determines whether the optimal schedule includes the j^{th} crash of seed i:

$$c_{i,j} = \begin{cases} 1 & \text{The schedule includes crash } j \text{ for seed } i \\ 0 & \text{otherwise} \end{cases}$$

Note that multiple crashes $c_{i,j}$ may correspond to the same bug. Crashes are triaged to unique bugs via a uniqueness function denoted by μ. In our experiments, we use stack hash [24], a non-perfect but industry standard method. Thus, if the total number of unique stack hashes is U, we say we found U unique bugs in total. The invariant is:

$$b_x = 1 \text{ iff } \exists\, i, j : \mu(c_{i,j}) = x \tag{1}$$

Thus, if two crashes $c_{i,j}$ and $c_{i',j'}$ have the same hash, a schedule can get at most one unique bug by including either or both crashes.

Finally, we include a cost for finding each bug. We associate with each crash the incremental fuzzing cost for seed S_i to find the bug:

$$\forall i : t_{i,j} = \begin{cases} a_{i,1} & , j = 1 \\ a_{i,j} - a_{i,j-1} & , j > 1 \end{cases}$$

where $a_{i,j}$ is the arrival time for the $c_{i,j}$ crash, and $t_{i,j}$ represents interarrival time—the time interval between the occurrences of $c_{i,j-1}$ and $c_{i,j}$. Figure 2 visually illustrates the connection between $c_{i,j}$, b_x and $t_{i,j}$.

We are now ready to phrase optimal scheduling with a fixed time-budget as an ILP maximization problem:

$$\begin{aligned}
&\text{maximize} && \sum_x b_x \\
&\text{subject to} && \bigwedge_{i,j} . \, c_{i,j+1} \leq c_{i,j} && (2) \\
& && \sum_{i,j} c_{i,j} \cdot t_{i,j} \leq t_{\text{thres}} && (3) \\
& && \bigwedge_{i,j} . \, c_{i,j} \leq b_x \text{ where } \mu(c_{i,j}) = x && (4) \\
& && \bigwedge_{x} . \, b_x \leq \sum_{i,j} c_{i,j} \text{ where } \mu(c_{i,j}) = x && (5)
\end{aligned}$$

Constraint (2) ensures that the schedule considers the order of crashes found. In particular, if the j-th crash of a seed is found, all the previous crashes must be found as well. Constraint (3) ensures that the time to find all the crashes does not exceed our time budget t_{thres}. Constraints (4) and (5) link crashes and unique bugs. Constraints (4) says that if a crash is found, its corresponding bug (based on stack-hash) is found, and the next equation guarantees that if a bug is found, at least one crash triggering this bug was found.

Additionally, by imposing one extra inequality:

$$\sum_i c_{i,1} \leq k \tag{6}$$

we can bound the number of used seeds by k (if the first crash of a seed is not found, there is no value in fuzzing the seed at all), thus getting k-bounded optimal budgeted scheduling, which gives us the number of bugs found with the optimal minset of size up to k.

Optimal Seed Selection for Round-Robin. The formulation for optimal budgeted scheduling gives us a best solution any scheduling algorithm could hope to achieve both in terms of seeds to select (minset) and interleaving between explored seeds (scheduling). We can also model the optimal seed selection for specific scheduling algorithms with the ILP formulation. We show below how this can be achieved for Round-Robin, as this may be of independent interest.

Round-Robin scheduling splits the time budget between the seeds equally. Given a time threshold t_{thres} and N seeds, each seed will be fuzzed for $\frac{t_{\text{thres}}}{N}$ units of time. Round-Robin is a simple but effective scheduling algorithm in many adversarial scenarios [29]. Simulating Round-Robin for a given set of seeds is straightforward, but computing the optimal subset of seeds of size k with Round-Robin cannot be solved with a polynomial algorithm. To obtain the optimal minset for Round-Robin, we add the following inequality to Inequalities 2-6:

$$\forall_i . \sum_j c_{i,j} \cdot t_{i,j} \leq \frac{t_{\text{thres}}}{k} \tag{7}$$

The above inequality ensures that none of the seeds will be explored for more than $\frac{t_{\text{thres}}}{k}$ time units, thus guaranteeing that our solution will satisfy the Round-Robin constraints. Similar extensions can be used to obtain optimal minsets for other scheduling algorithms.

4 Q3: Transferability of Seed Files

Precomputing a good seed set for a single application P_1 may be time intensive. For example, the first step in a minset-based approach is to run each seed dynamically to collect coverage information. Collecting this information may not be cheap. Collecting coverage data often requires running the program in a dynamic analysis environment like PIN [18] or Valgrind [25], which can slow down execution by several orders of magnitude. In our own experiments, collecting coverage information on our data set took 7 hours. One way COVERSET could minimize overall cost is to find a "good" set of seeds and reuse them from one application to another.

There are several reasons to believe this may work. One reason is most programs rely on only a few libraries for PDF, image, and text processing. For example, if application P_1 and P_2 both link against the `poppler` PDF library, both applications will likely crash on the same inputs. However, shared libraries are typically easy to detect, and such cases may be uninteresting. Suppose instead P_1 and P_2 both have independent implementations, e.g., P_1 uses `poppler` and P_2 uses the GhostScript graphics library. One reason P_1 and P_2 may crash on similar PDFs is that there are intrinsically hard portions of the PDF standard to implement right, thus both are likely to get it wrong. However, one could speculate any number of reasons the bugs in applications would be independent. To the best of our knowledge, there has been no previous systematic investigation to resolve this question when the bugs are found via fuzzing.

5 System Design

A precondition to fuzzing is configuring the fuzzer to take the seed file as input. In this step, we are given the entire database of seeds and a particular program under test P. To fuzz, we must:

1. Recover P's command line options.
2. Determine which argument(s) causes P to read in from the fuzzing source. For simplicity, we focus on reading from a file, but the general approach may work with other fuzzing sources.
3. Determine the proper file type, e.g., giving a PDF reader a PDF as a seed is likely to work better than giving a GIF. We say a file type is valid for a program if it does non-trivial processing on the file.

Current fuzz campaigns typically require a human to specify the above values. For our work, we propose a set of heuristics to help automate the above procedure.

In our approach, we first use simple heuristics to infer likely command lines. The heuristics try the obvious and common command line arguments for accepting files, e.g., `-f file`. We also brute force common help options (e.g., `-help`) and parse the output for additional possible command line arguments.

As we recover command lines, we check if they cause P to read from a file as follows. We create a unique file name x, run P `x`, and monitor for system calls that open `x`.

In our data set we have 274 different file types, such as JPEG, GIF, PNG, and video files. Once we know the proper way to run P with seeds, the question becomes which seed types should we give P? We infer appropriate file types based on the following hypothesis: if s is a seed

handled by P and s' is not, then we expect the coverage of $P(s) > P(s')$. This hypothesis suggests an efficient way to infer file types accepted by an application. First, create a set of sample file type seeds F, where each element consists of a seed for a unique file type. Second, for each $s_i \in F$, count the number of basic blocks executed by $P(s_i)$. Third, select the top candidate (or candidates if desired) by total execution blocks. Though simple, we show in our evaluation this strategy works well in practice.

6 Experiments

We now present our experiments for checking the validity of the hypotheses introduced in § 2.2 and evaluate the overall performance in terms of bug discovered of COVERSET. We start by describing our experimental setup.

Experimental Setup. All of our experiments were run on medium and small VM instance types on Amazon EC2 (the type of the instance used is mentioned in every experiment). All VMs were running the same operating system, Debian Linux 7.4. The fuzzer used throughout our experiments is the CERT Basic Fuzzing Framework (BFF) [15]. All seed files gathered for our fuzzing experiments (4,912,142 files making up more than 6TB of data) were automatically crawled from the internet using the Bing API. Specifically, file type information was extracted from the open source Gnome Desktop application launcher data files and passed to the Bing API such that files of each type could be downloaded, filtered, and stored on Amazon S3. Coverage data was gathered by instrumenting applications using the Intel PIN framework and a standard block-based coverage collection PIN tool.

6.1 Establishing Ground Truth

To test the MINSET hypotheses, we need to obtain the ground truth (recall from § 3.1) for a fuzzing campaign that accounts for every possible seed selection and scheduling. We now present our methodology for selecting the target applications, files to fuzz, and parameters for computing the ground truth.

Target Applications. We selected 10 applications and 5 popular file formats: PDF, MP3, GIF, JPG and PNG for our experiments. Our program selection contains GUI and command line applications, media viewers, players, and converters. We manually mapped each program to a file format it accepts and formed 13 distinct (application, file formats) to be fuzzed—shown in Table 2. We selected at least two distinct command lines for each file type to test transferability (Hypothesis 3).

Seed Files. For each file type used by the target applications, we sampled uniformly at random 100 seed files (hence selecting $|\mathbb{F}| = 100$ for the seed file pool size) of the corresponding type from our seed file database. Note that determining ground truth for a single seed requires 12 hours, thus finding ground truth on all 4,912,142 is—for our resources—infeasible.

Fuzzing Parameters. Each of the target applications was fuzzed for 12 hours with each of the 100 randomly selected seed files of the right file type. Thus, each target application was fuzzed for 1,200 hours for a total of 650 CPU-days on an EC2 (m1.small) instance. All detected crashes were logged with timestamps and triaged based on BFF's stack hash algorithm.

The end result of our ground truth experiment is a log of crashes for each (seed file, application) tuple:

$$BFF(\{S_i\}, t_{\text{thres}} = 12h) = \{(b_1, S_i, t_1), \ldots, (b_k, S_i, t_k)\}$$

Fuzzing results. BFF found 2,941 unique crashes, identified by their stack hash. BFF crashed 8 programs out of the 10 target applications. 2,702 of the unique crashes were found on one application, `mp3gain`. Manual inspection showed that the crashes were due to a single exploitable buffer overflow vulnerability that mangled the stack and confused BFF's stack-based uniqueness algorithm. When reporting our results, we therefore count the 2,702 unique crashes in `mp3gain` as one. With that adjustment, BFF found 240 bugs. Developing and experimenting with more robust, effective, and accurate triaging algorithms is an open research problem and a possible direction for future work.

Simulation. The parameters of the experiment allow us to run simulations and reason about all possible seed selections (among the 100 seeds of the application) and scheduling algorithms for a horizon of 12 hours on a single CPU. Our simulator uses our ILP formulation from § 3 to compute optimal seed selections and scheduling for a given time budget. Using the ground truth, we can run simulations to evaluate the performance of hour-long fuzzing campaigns within minutes, following a replay-based fuzzing simulation strategy similar to FUZZSIM [29].

We used the simulator and ran a set of experiments to answer the following three questions: 1) how good are seed selection algorithms when compared against RANDOM SET (§ 6.2) and when compared against each other (§ 6.2.1)?, 2) can we reuse reduced sets across programs (§ 6.3)?, and 3) can our algorithm correctly identify file types for applications (§ 6.4)?

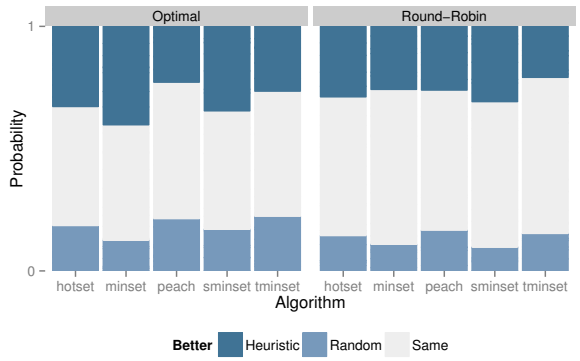


Figure 3: Comparing bug-finding performance of seed selection algorithms against RANDOM SET.

	Optimal	Round-Robin
HOT SET	63.58%	67.12%
PEACH SET	50.64%	60.30%
UNWEIGHTED MINSET	75.24%	70.24%
SIZE MINSET	66.33%	75.78%
TIME MINSET	52.60%	57.62%

Table 1: Conditional probability of an algorithm outperforming RANDOM SET with k=10, given that they do not have the same performance (P_{win}).

6.2 Are Seed Selection Algorithms Better than Random Sampling?

Spending resources on a seed selection algorithm is only useful if the selected seeds outperform random seed sampling (RANDOM SET). In this experiment, we compare the performance of selection algorithms as presented in §2.1 against the random sampling baseline.

All selection algorithms are deterministic, while RANDOM SET is randomized. Thus, we cannot show that RANDOM SET is always better (or worse), but we can instead compute the probability that RANDOM SET is better (or worse). To estimate the probability, we setup the following random experiment: we randomly sample a set of seeds—the size of the set is the same ($k = 10$ in our experiment for an order of magnitude reduction) as the competing reduced set—from the seed pool and measure the number of bugs found. The experiment has three possible outcomes: 1) the random set finds more bugs, 2) the random set finds fewer bugs, or 3) the random and the competitive set find the same number of bugs.

We performed 13,000 repetitions of the above experiment—1,000 for each (application, file format) tuple—and measured the frequency of each event when the optimal scheduling algorithm is employed for both. We then repeated the same experiment while using Round-Robin as the scheduling algorithm. We calculated the probability by dividing the frequency by the number of samples. Figure 3 summarizes the results. For instance, the left-most bar is the result for HOT SET with the optimal scheduling. You can see that HOT SET finds more bugs than a RANDOM SET of the same size with a probability of 32.76%, and it is worse with a probability of 18.57%. They find the same amount of bugs with a probability of 48.66%.

The first pattern that seems to persist through scheduling and selection algorithms (based on Figure 3) is that there is a substantial number of ties—RANDOM SET seems to behave as well as selection algorithms for the majority of the experiments. This is not surprising, since 3/13 (23%) of our (application, file format) combinations—(mplayer, MP3), (eog, JPG), (jpegtran, JPG)—do not crash at all. With no crash to find, any algorithm will be as good as random. Thus, to compare an algorithm to RANDOM SET we focus on the cases where the two algorithms differ, i.e., we compute the conditional probability of winning when the two algorithms are not finding the same number of bugs.

We use P_{win} to denote the conditional probability of an algorithm outperforming RANDOM SET, given that they do not have the same performance. For example, for SIZE MINSET, P_{win} is defined as: $P[\text{SIZE MINSET} > \text{RANDOM SET} \mid \text{SIZE MINSET} \neq \text{RANDOM SET}]$. Table 1 shows the values of P_{win} for all algorithms for sets of size $k = 10$. We see that UNWEIGHTED MINSET and SIZE MINSET are the algorithms that more consistently outperform RANDOM SET with a P_{win} ranging from 66.33% to 75.78%. HOT SET immediately follows in the 63-67% range, and TIME MINSET, PEACH SET have the worst performance. Note that PEACH SET has a P_{win} of 50.64% in the optimal schedule effectively meaning that it performs very close to a random sample on our dataset.

Conclusion: seed selection algorithms help. With the exception of the PEACH SET and TIME MINSET algorithms which perform very close to RANDOM SET, our data shows that heuristics employed by seed selection algorithms perform better than fully random sampling. Thus, hypothesis 1 seems to hold. However, the bug difference is not sufficient to show that *any* of the selection algorithms is *strictly* better with statistical significance. Fuzzing for longer and/or obtaining the ground truth for a larger seed pool are possible future directions for showing that seed selection algorithms are *strictly* better than choosing at random.

Files	Programs	Crashes	Bugs	Random Set		Hot Set		Unweighted Minset		Time Minset		Size Minset		Peach Set	
				#S	#B	#S	#B	#S	#B	#S	#B	#S	#B	#S	#B
PDF	xpdf	706	57	10	7	10	9	32	19	32	16	40	19	54	31
	mupdf	6,570	88	10	13	10	14	40	29	43	29	49	31	59	31
	pdf2svg	5,720	81	10	14	10	27	36	48	39	43	45	47	53	49
MP3	ffmpeg	1	1	10	0	10	1	11	0	11	0	22	0	19	0
	mplayer	0	0	10	0	10	0	10	0	12	0	14	0	23	0
	mp3gain	434,400	2,702	10	92	10	9	9	150	8	74	10	74	14	175
GIF	eog	9	1	10	0	10	1	29	0	27	0	43	1	44	1
	convert	72	2	10	1	10	1	13	1	14	0	24	2	22	1
	gif2png	162,302	6	10	4	10	4	16	5	17	5	29	5	33	4
JPG	eog	0	0	10	0	10	0	31	0	31	0	47	0	53	0
	jpegtran	0	0	10	0	10	0	10	0	12	0	21	0	23	0
PNG	eog	123	2	10	1	10	1	30	2	30	2	45	2	49	2
	convert	2	1	10	0	10	0	11	1	12	1	17	1	16	1
Total		609,905	2,941		132		67	278	255	288	170	406	182	462	295

Table 2: Programs fuzzed to evaluate seed selection strategies and obtain ground truth. The columns include the number of seed files (#S) obtained with each algorithm, and the number of bugs found (#B) with the optimal scheduling strategy.

6.2.1 Which Algorithm Performed Best?

Table 2 shows the full breakdown of the reduced sets computed by each algorithm with the optimal scheduling algorithm. Columns 1 and 2 show the file type and program we are analyzing, while columns 3 and 4 show the total number of crashes and unique bugs (identified by stack hash) found during the ground truth experiment. The next six columns show two main statistics (in subcolumns) for each of the seed selection algorithms: 1) the size of the set k (#S), and 2) the number of bugs (#B) identified with optimal scheduling. All set cover algorithms (Peach Set, Unweighted Minset, Time Minset, Size Minset) were allowed to compute a full-cover, i.e., select as many files as required to cover all blocks. The other two algorithms (Random Set and Hot Set) were restricted to sets of size $k = 10$.

Bug Distribution and Exploitability. The fuzzing campaign found bugs in 10/13 configurations of ⟨program, file type⟩, as shown in table 2. In 9/10 configurations we found less than 100 bugs, with one exception: `mp3gain`. We investigated the outlier further, and discovered that our fuzzing campaign identified an exploitable stack overflow vulnerability—the mangled stack trace can create duplicates in the stack hash algorithm. We verified the bug is exploitable and notified the developers, who promptly fixed the issue.

Reduced Set Size. Table 2 reflects the ability of the set cover algorithms to reduce the original dataset of 100 files. As expected, Unweighted Minset is the best in terms of reduction ability, with 278 files for obtaining full cover. Time Minset requires slightly more files (288). Size Minset and Peach Set require almost twice as many files to obtain full cover (406 and 462 respectively).

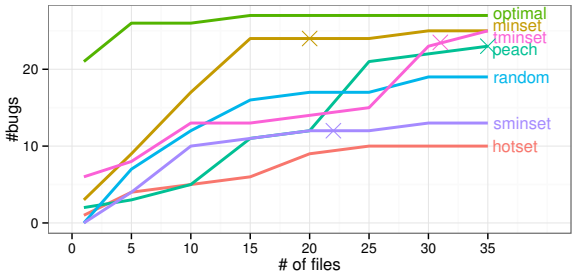


Figure 4: Number of bugs found by different seed selection algorithms with optimal scheduling.

Bug Finding. The Peach Set algorithm finds the highest number of bugs (295), followed by Unweighted Minset (255), Size Minset (182) and Time Minset (170). Hot Set and Random Set find substantially fewer bugs when restricted to subsets of size up to 10. We emphasize again that bug counts are measured under optimal scheduling and thus size of the reduced set is analogous to the performance of the selection algorithm (the highest number of bugs will be found when all seeds are selected). Thus, to compare sets of seeds in terms of bug-finding ability we need a head to head comparison where sets have the same size k.

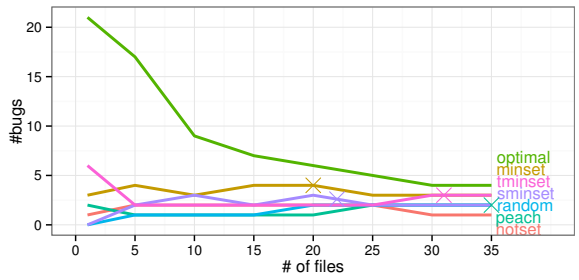


Figure 5: Number of bugs found by different seed selection algorithms with Round-Robin.

Figure 4 shows all selection algorithms and how they perform in terms of average number of bugs found as a function of the parameter k—the size of the seed file set. The "×" symbols represent the size after which each algorithm achieves a full cover (after that point extra files are added sorted by the metric of the selection algorithm, e,g,, by coverage in UNWEIGHTED MINSET). As witnessed in the comparison against RANDOM SET, UNWEIGHTED MINSET consistently performs better than other seed selection algorithms. TIME MINSET and PEACH SET also eventually converge to the performance of UNWEIGHTED MINSET under optimal scheduling, closely followed by random. HOT SET performs the worst, showing that spending time exploring *all* seeds can be wasteful. We also note, that after obtaining full cover (at 20 seed files), UNWEIGHTED MINSET's performance does not improve at the same rate—showing that adding new files that do not increase code coverage is not beneficial (even with optimal scheduling).

We performed an additional simulation, where all reduced sets were run with Round-Robin as the scheduling algorithm. Figure 5 shows the performance of each algorithm as a function of the parameter k. Again, we notice that that UNWEIGHTED MINSET is outperforming the other algorithms. More interestingly, we also note that UNWEIGHTED MINSET's performance actually *drops* after obtaining full cover. This shows that minimizing the number of seeds is important; adding more seeds in Round-Robin seems to hurt performance for all algorithms.

Conclusion: UNWEIGHTED MINSET performed best. UNWEIGHTED MINSET outperformed all other algorithms in our experiments, both for optimal and Round-Robin scheduling. This experiment confirms conventional wisdom that suggests collecting seeds with good coverage for successful fuzzing. More importantly, it also shows that computing a minimal cover with an approximation with a proven competitiveness ratio (UNWEIGHTED MINSET) is better than using an algorithm with no guaranteed competitive ratio (PEACH SET).

File	Application	FULL SET	UNWEIGHTED MINSET (k=10)
PDF	xpdf	53%	70%
	mupdf	83%	90%
	pdf2svg	71%	80%
MP3	ffmpeg	1%	0%
	mplayer	0%	0%
	mp3gain	95%	100%
GIF	eog	8%	0%
	convert	12%	10%
	gif2png	97%	100%
JPG	eog	0%	0%
	jpegtran	0%	0%
PNG	eog	22%	30%
	convert	2%	10%

Table 3: Probability that a seed will produce a bug in 12 hours of fuzzing.

6.2.2 Are reduced seed sets better than a full set?

Hypothesis 4 tests the premise of using a reduced data set. Will a reduced set of seeds find more bugs than the full set? We simulated a fuzzing campaign with the full set, and with different reduced sets. We compare the number of bugs found by each technique.

Using the optimal scheduling, the full set will *always* find more, or the same amount of bugs, than any subsets of seeds. Indeed, the potential schedules of the full set is a superset of the potentiel schedules of any reduced set. The optimal schedule of a reduced set of seeds is a valid schedule of the full set, but the optimal schedule of the full set might not be a valid schedule of a reduced set. Hypothesis 4 is therefore false under the optimal scheduling. We use a Round-Robin schedule to answer this question more realistically.

The "×" symbols on Figure 5 shows the unpadded size of the different selection algorithms. For those sizes, UNWEIGHTED MINSET found 4 bugs on average, and the other MINSET algorithms found between 2.5 and 3 bugs. Fuzzing with the full set uncovered only 1 unique bug on average.

We also measure the quality of a set of seeds by looking at the average seed quality contained in that set. Our hypothesis is that a reduced set increases the average seed quality compared to the full set. To measure quality, we computed the probability of a seed producing a bug after fuzzing it for 12 hours, when the seed is picked from the full set or the UNWEIGHTED MINSET. Table 3 lists the

results of this experiment. The UNWEIGHTED MINSET had a higher seed quality than the full set in 7 cases, while the opposite was true in 3 cases. They were tied on the 3 remaining cases.

Conclusion: Fuzzing with a reduced sets is more efficient in practice. The UNWEIGHTED MINSET outperformed the full set in our two experiments. Our data demonstrates that using seed selection techniques is beneficial to fuzzing campaigns.

6.3 Are Reduced Sets Reusable Across Programs?

We showed that seed selection algorithms improve fuzzing in terms of bug-finding performance. However, performing the data reduction may be computationally expensive; for instance, all set cover algorithms require collecting coverage information for all the seeds. Is it more profitable to invest time computing the minset to fuzz an efficient reduced set, or to simply fuzz the full set of seeds for the full time budget? In other words, is the seed selection worth the effort to be performed online?

We answer that question by presenting parts of our dataset. For example, our JPG bucket contains 530,727 distinct files crawled from the web. Our PIN tool requires 55 seconds (on average based on the 10 applications listed in Table 2) to compute code coverage for a single seed. Collecting coverage statistics for all our JPG files would take 368 CPU-days. For fuzzing campaigns shorter than a year, there would not be enough time to compute code coverage, let alone finding more bugs than the full set.

The result above indicates that, while seed selection techniques help improve the performance of fuzzing, their benefits may not outweigh the costs. It is impractical to spend a CPU year of computation to perform a separate seed selection for every new application that needs fuzzing, thus indicating that Hypothesis 2 does not hold.

However, recomputing the reduced set for *every application* may not be necessary. Instead, we can compute a reduced set for every *file type*. Our intuition is a reduced set that is of high-quality for application A should also be high-quality for application B—assuming they accept the same file type. Thus, precomputing reduced sets for popular file types *once*, would allow us to instantly select a high-quality set of seed files to start fuzzing. To test transferability of reduced sets (Hypothesis 3), we measure seed quality by computing code coverage achieved by a MINSET across programs.

Do Reduced Sets Transfer Coverage? Using the seed files from our ground truth experiment (§ 6.1) we measured the cumulative code coverage achieved in each configuration (program and file format) with reduced UNWEIGHTED MINSETs computed on all other configurations (for a total of $13 \times 13 \times 100$ coverage measurements). All measurements were performed on a c1.medium instance on amazon.

Figure 6 is a heat map summarizing our results. The configurations on the bottom (x-axis) represent all computed UNWEIGHTED MINSETs, while the configurations on the left (y-axis) represent the configurations tested. Darker colors indicate that the selected UNWEIGHTED MINSET obtains higher coverage. For example, if we select the `pdf.mupdf` MINSET from the x-axis, we can see how it performs on all the other configurations on the y-axis. For instance, we notice that `pdf.mupdf` MINSET performs noticeably better on 5 configurations: `pdf.mupdf` (expected since this is the configuration on which we computed the MINSET), `pdf.xpdf` and `pdf.pdf2svg` (expected since these applications also accept `pdfs`), and interestingly `png.convert` and `gif.convert`. Initially we were surprised that a PDF MINSET would perform so well on `convert`; it turns out that this result is not surprising since `convert` can also process PDF files. Similar patterns can be similarly explained—for example, GIF MINSETs are performing *better* than MP3 MINSETs for `mplayer`, simply because `mplayer` can render GIF images.

The heat map allows us to see two clear patterns:

1. High coverage indicates the application accepts a file type. For instance, by following the row of the `gif.eog` configuration we can immediately see that `eog` accepts GIF, JPG, and PNG files, while it does not process MP3s or PDFs. This is exactly the same pattern we are exploiting in our file type inference algorithm (§ 6.4).

2. Coverage transfers across applications that process the same file type. For example, we clearly see the PDF cluster forming across all PDF configurations, despite differences in implementations. While `xpdf` and `pdf2svg` both use the `poppler` library for processing PDFs, `mupdf` has a completely independent implementation. Nevertheless, `mupdf`'s MINSET performs well on `xpdf` and vice versa. Our data shows that similar clusters appear throughout configurations of the same file type, suggesting that we can reuse MINSETs across applications that accept the same file type (Hypothesis 3).

Conclusion: Reduced sets are transferable. Our data suggests that reduced sets can be transferred to programs parsing the same file types with respect to code coverage. Therefore, it is necessary to compute only one reduced set per file type.

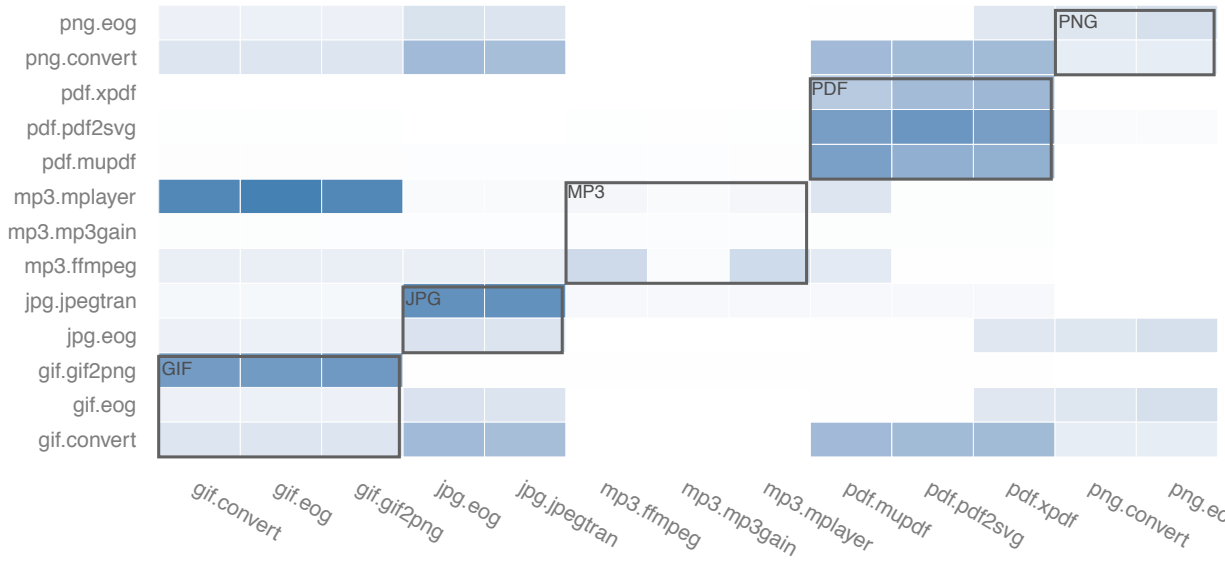


Figure 6: Transferability of UNWEIGHTED MINSET coverage across configurations. The base configurations, on which the reduced sets were computed, are on the bottom; the tested configurations are on the left. Darker colors indicate higher coverage.

6.4 Inferring File Types

In this experiment we ran our algorithm to automatically infer file types for our 10 applications over a large body of diverse file types (88 in total including AVI, MP3, MKV and so forth). We then manually verified the inferred file type, and measured accuracy. We report an error if a reported file type is not recognized by the target application. Table 6.4 summarizes our results. Our file type inference algorithm successfully infers file types for every program except `mp3gain`, where the file type inferred was CBR (Comic Book Reader), instead of MP3.

We manually examined why `mp3gain` shows higher code coverage for CBR files. It turns out that our sample CBR file is larger than 15 MB, and it happens to have a valid MP3 frame header signature in the middle of the file. Since `mp3gain` searches for a valid MP3 frame header regardless of the entire file format, it is possible to misinterpret an input file as a valid MP3 file. In other words, this is probably not a false positive, because `mp3gain` indeed takes in the CBR file and outputs a modified CBR file, which is the expected behavior of the program.

Program	Inferred File Type	Success
convert	svg	✓
eog	png	✓
ffmpeg	divx	✓
gif2png	gif	✓
jpegtran	jpeg	✓
mp3gain	cbr	✗
mplayer	avi	✓
mupdf	pdf	✓
pdf2svg	pdf	✓
xpdf	pdf	✓

Table 4: File-type inference results on our dataset.

7 Discussion & Future Work

Input Types. The focus of this paper is on file-parsing applications. Thus, we do not target applications that use command line arguments as their input sources (e.g., /bin/echo), or applications that receive input from the network (e.g., /usr/bin/wget). File-based vulnerabilities represent a significant attack vector, since remote attacks can be carried out by simply sending an attachment to the victim over the network.

Handling argument inputs for applications is straightforward: we can extend our fuzzing framework to randomly generate arguments to the `exec` system call. Treating network applications would be more elaborate since we would have to update our seed database to include network packets for various protocol types. One potential extension is to utilize automatic protocol reversing [3, 7]. We leave it as future work to support more input types.

Statistical Significance. We have performed initial hypothesis testing based on the data. Currently, using UNWEIGHTED MINSET is assumed to outperform other al-

gorithms. Using our data (see Figure 3), we were able to show that UNWEIGHTED MINSET is at least as good as random with high probability. However, the data does not show with statistical significance that UNWEIGHTED MINSET is strictly better. Fuzzing longer and with more seed files and programs may yield more datapoints that would allow a stronger conclusion—at the cost of a much more costly ground truth computation: an additional seed file requires 12 hours of extra fuzzing hours for each application. We leave fuzzing for more than 650 days as future work.

Command Line Inference. Currently, COVERSET uses several heuristics to infer command line arguments. We believe that command line inference is an important first step to fully automate the entire fuzzing process. For example, COVERSET currently does not handle dependent arguments, e.g., when option `A` is valid only when option `B` is also selected. Developing systematic and effective approaches for deriving command line arguments—e.g., based on white-box techniques—is a possible direction for future work.

8 Related Work

In early 90s, Miller et al. [21] introduced the term fuzzing. Since then, it has become one of the most widely-deployed technique for finding bugs. There are two major categories in fuzzing based on its ability to examine the internal of the software under test: (1) black-box fuzzing [20], and (2) white-box fuzzing [13, 14]. In this paper, we use black-box mutational fuzzing as the underlying technique for our data reduction algorithms.

Coverage-driven seed selection is not new. Several papers and security practitioners use similar heuristics to select seeds for fuzzing [1, 8, 9, 11, 22, 23, 26]. FuzzSim by Woo et al. [29] is the closest work from academia, where they tackle a seed scheduling problem using multi-armed bandit algorithms. Our paper differs from their approach in that we are not developing an online scheduling algorithm, but an offline data-driven seed selection approach. Therefore, our seed selection is complementary—when it is used as a preprocessing step—to the FuzzSim scheduling algorithm.

There are several previous works on recovering input formats, which involves dynamic taint analysis [3, 7]. Our goal is being able to run fuzzing with appropriate seed files, but not recovering the semantics of file format. Accommodating more precise file format inference techniques is out of the scope of this paper.

9 Conclusion

In this paper we designed and evaluated six seed selection techniques. In addition, we formulated the optimal ex post facto seed selection scheduling problem as an integer linear programming problem to measure the quality of seed selection algorithms. We performed over 650 days worth of fuzzing to determine ground truth values and evaluated each algorithm. We found 240 new bugs. Our results suggest how best to use seed selection algorithms to maximize the number of bugs found.

Acknowledgments

We would like to thank Alan Hall, and our anonymous reviewers for their comments and suggestions. This work is supported in part by the NSF-CNS0953751, DARPA CSSG-FA9750-10-C-0170, and the SEI-FA8721-05-C-0003. This work reflects only the opinions of the authors, not the sponsors.

References

[1] ABDELNUR, H., LUCANGELI, O., AND FESTOR, O. Spectral fuzzing: Evaluation & feedback.

[2] ARIANE. The ariane catastrophe. http://www.around.com/ariane.html.

[3] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer & Communications Security* (2007).

[4] CHVATAL, V. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research 4*, 3 (1979), 233–235.

[5] CNN. Toyota recall costs: $2 billion. http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm, 2010.

[6] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms , Second Edition*, vol. 7. 2001.

[7] CUI, W., PEINADO, M., AND CHEN, K. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008).

[8] DURAN, D., WESTON, D., AND MILLER, M. Targeted taint driven fuzzing using software metrics. In *CanSecWest* (2011).

[9] EDDINGTON, M. Peach fuzzer. http://peachfuzzer.com/.

[10] FEIGE, U. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM 45*, 4 (1998), 634–652.

[11] FRENCH, T., AND PROJECT, V. Closed loop fuzzing algorithms.

[12] GARTNER. Software security market at 19.2 billion. http://www.gartner.com/newsroom/id/2500115, 2012.

[13] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium* (2008), no. July.

[14] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: Whitebox fuzzing for security testing. *Communications of the ACM 55*, 3 (2012), 40–44.

[15] HOUSEHOLDER, A. D., AND FOOTE, J. M. Probability-based parameter selection for black-box fuzz testing. Tech. Rep. August, CERT, 2012.

[16] JOHNSON, D. S. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences 9* (1974), 256–278.

[17] LANGNER, R. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy Magazine 9*, 3 (May 2011), 49–51.

[18] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tols with dynamic instrumentation. In *Programming Language Design and Implementation* (2005), ACM, pp. 190–200.

[19] MARKET RESEARCH MEDIA. U.s. federal cybersecurity market forecast 2013-2018. http://www.marketresearchmedia.com/?p=206, 2013.

[20] MCNALLY, R., YIU, K., AND GROVE, D. Fuzzing : The state of the art.

[21] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Communications of the ACM 33*, 12 (1990), 32–44.

[22] MILLER, C. Fuzz by number. In *CanSecWest* (2008).

[23] MILLER, C. Babysitting an army of monkeys. In *CanSecWest* (2010).

[24] MOLNAR, D., LI, X., AND WAGNER, D. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium* (2009), pp. 67–82.

[25] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89*, 2 (Oct. 2003), 44–66.

[26] OPSTAD, L., AND MOLNAR, D. Effective fuzzing strategies. Tech. rep., 2010.

[27] TEAM, C. S. Clusterfuzz. https://code.google.com/p/clusterfuzz/.

[28] UHLEY, P. A basic distributed fuzzing framework for foe. https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html.

[29] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM Conference on Computer & Communications Security* (2013), pp. 511–522.

LibFTE: A Toolkit for Constructing Practical, Format-Abiding Encryption Schemes

Daniel Luchaup[1] *luchaup@cs.wisc.edu*
Kevin P. Dyer[2] *kdyer@cs.pdx.edu*
Somesh Jha[1] *jha@cs.wisc.edu*
Thomas Ristenpart[1] *rist@cs.wisc.edu*
Thomas Shrimpton[2] *teshrim@cs.pdx.edu*

[1]*Department of Computer Sciences, University of Wisconsin-Madison*
[2]*Department of Computer Science, Portland State University*

Abstract

Encryption schemes where the ciphertext must abide by a specified format have diverse applications, ranging from in-place encryption in databases to per-message encryption of network traffic for censorship circumvention. Despite this, a unifying framework for deploying such encryption schemes has not been developed. One consequence of this is that current schemes are ad-hoc; another is a requirement for expert knowledge that can disuade one from using encryption at all.

We present a general-purpose library (called libfte) that aids engineers in the development and deployment of format-preserving encryption (FPE) and format-transforming encryption (FTE) schemes. It incorporates a new algorithmic approach for performing FPE/FTE using the nondeterministic finite-state automata (NFA) representation of a regular expression when specifying formats. This approach was previously considered unworkable, and our approach closes this open problem. We evaluate libfte and show that, compared to other encryption solutions, it introduces negligible latency overhead, and can *decrease* diskspace usage by as much as 62.5% when used for simultaneous encryption and compression in a PostgreSQL database (both relative to conventional encryption mechanisms). In the censorship circumvention setting we show that, using regular-expression formats lifted from the Snort IDS, libfte can reduce client/server memory requirements by as much as 30%.

1 Introduction

Both in practice and in the academic literature, we see an increasing number of applications demanding encryption schemes whose ciphertexts abide by specific formatting requirements. A small industry has emerged around the need for in-place encryption of credit-card numbers, and other personal and financial data. In the case of credit-card numbers, this means taking in a string of 16 decimal digits as plaintext and returning a string of 16 decimal digits as ciphertext. This is an example of *format-preserving encryption* (FPE). NIST is now considering proposals for standardized FPE schemes, such as the FFX mode-of-operation [7], which is already used in some commercial settings [3]. On a totally different front, a recent paper [11] builds a *format-transforming encryption* scheme. It takes in plaintext bit strings (formatted or not) and returns ciphertexts formatted to be indistinguishable, from the point of view of several state-of-the-art network monitoring tools, from real HTTP, SMTP, SMB or other network protocol messages. This FTE scheme is now part of the Tor Project's Browser Bundle, and is being integrated into other anti-censorship systems.

It seems clear that FPE and FTE have great potential for other applications, too. Unfortunately, developers will find a daunting collection of design choices and engineering challenges when they try to use existing FPE or FTE schemes in new applications, or to instantiate entirely new schemes. To begin with, there isn't a standard way to specify the formats that plaintexts or ciphertexts must respect. There are no established guidelines, and certainly no automated tools, to help developers understand whether they should be targeting deterministic schemes or randomized ones, or how their chosen formats might affect runtime performance and memory usage. (In the case of FTE, it can be difficult to tell if a given input and output format will result in a scheme that operates properly.) There are no established APIs, and no reference implementations or open-source libraries to aid development.

Making FPE/FTE More Approachable: libfte. In this work, we offer a unifying framework for building and deploying FPE and FTE schemes. We design and implement an algorithm library, libfte, and include in it developer-assistance tools. A paramount goal of

our effort is *ease-of-use*: our library exposes an interface in which formats for plaintexts and ciphertexts are easily specified via Perl-compliant regular expressions (regexes), and it relieves the programmer of the burdens of making good algorithm and parameter choices.

Some of what we do is to make existing algorithms (e.g., FFX) significantly easier to use. But some of the engineering and deployment challenges demand entirely new approaches to both FPE and FTE. Perhaps most notably, we solve an open problem regarding how to build regular-expression-based schemes using a regex's nondeterministic finite automaton (NFA) representation, as opposed to its DFA representation. This is desirable because it can lead to significantly more space-efficient schemes, but the approach was previously thought to be unworkable [5, 11]. We dispel this thought, and experimentally observe the resulting boost in efficiency.

To summarize the main contributions of this work, we:

- *Design and implement a library and toolkit* to make development and deployment easy. The libfte library exposes simple interfaces for performing FPE/FTE over regex formats specified by the user. We provide a configuration tool that guides developers towards good choices for the algorithms that will instantiate the scheme, and that provides concrete feedback on expected offline and online performance and memory usage.
- *Develop new FTE schemes* that take regular-expression formats, but can work directly with their NFA representation. This was previously thought to be an unworkable approach [5], due to a PSPACE-hardness result, but we show how to side-step this via a new encoding primitive called *relaxed ranking*. The result is FTE schemes that handle a larger class of regexes, and impose smaller offline/online memory requirements.
- *Detail a general, theoretical framework* that captures existing FPE/FTE schemes as special cases, and surfaces potentially useful new constructions, e.g., deterministic FTE that encrypts and compresses simultaneously. Due to space constraints, the formalisms appear mostly in the full version [16].

In addition, the libfte library will be made publicly available as free and open-source software[1], with APIs for Python, C++ and JavaScript.

Applications. We exercise libfte by applying it to a variety of application settings. Table 1 gives a summary of the diversity of formats required across these various applications.

We first show how to use libfte to perform FPE of SQL database fields, a classic motivational setting for FPE, but one that has (to the best of our knowledge) never been reported upon. We show that performance loss compared to conventional encryption is negligible. We also show how to leverage the flexibility of libfte to *improve* performance, by using a (deterministic) FTE scheme that simultaneously encrypts and compresses fields (in a provably secure manner).

We then use libfte to build a proof-of-concept browser plugin that encrypts form data on websites such as Yahoo! Contacts. This uses a variety of FPE and FTE schemes, and allows one to abide by a variety of format restriction checks performed by the website.

Finally, we show that our NFA-based algorithms in libfte enable significant memory savings, specifically for the case of using FTE in the network-monitor-avoidance setting [11]. Using a corpus of 3,458 regular expressions from the Snort monitor we show that we can reduce memory consumption of this FTE application by 30%.

Deployment Setting	Examples	
	Type	Constraint
Databases	credit card number	16-digit string
	datefield	YYYYMMDD
	account balance	32-bit integers
Web Forms	email address	contains @ symbol, ends with {.com,...}
	year, month, day	YYYY, MM, DD
	URL	starts with http(s)
Network Monitors	HTTP GET request	"GET /..."
	Browser X	"...User-Agent: X ..."
	SSH traffic	"SSH-..."

Table 1: Example deployment settings and constraints for FPE/FTE schemes.

2 Previous Approaches and Challenges

We review in more detail some of the main results in the areas of format-preserving and format-transforming encryption, and then discuss some of the challenges presented when one attempts to implement and use these in practice. As we shall see, existing tools fall short for the types of applications we target. Table 2 provides a summary.

Format-preserving encryption. In many settings the format of a plaintext and its encryption must be the same, and the tool used to achieve this is format-preserving encryption (FPE). Work on FPE predates its name, with various special cases such as length-preserving encryption of bit strings for disk-sector encryption (c.f., [14, 15]), ciphers for integral sets [8], and elastic block ciphers [10] including de novo constructions such as the hasty pudding cipher [21]. For an overview of work on

[1] https://libfte.org/

Paper	Builds	Formats	Schemes	Implementation	Comments
[7]	FPE	slice of Σ^*	deterministic	none	proposed NIST standard
[5]	FPE	slice of chosen regular language	deterministic	none	first FPE paper, theory only, requires regex-to-DFA conversion
[11]	FTE	slice of chosen regular language	randomized	open source, but domain specific	input format fixed as bitstrings, control of output format, requires regex-to-DFA conversion
This Work	**FPE/ FTE**	**range-slice of chosen regular language**	**deterministic/ randomized**	**open source, configuration toolchain, non domain specific**	**control of input and output format, NFA and DFA ranking, regex-to-DFA conversion not required**

Table 2: Analysis of prior works, and a comparison of features.

FPE, see Rogaway [20].

FPE was first given a formal cryptographic treatment by Bellare, Ristenpart, Rogaway and Spies (BRRS) [5]. In their work, BRRS suggested an approach to FPE called the "rank-encipher-unrank" construction. First, they show how to build a cipher that maps $\mathbb{Z}_N$ to $\mathbb{Z}_N$, for an arbitrary fixed number N. (Recall that $\mathbb{Z}_N = \{0, 1, \ldots N-1\}$.) Now say that X is a set of strings that all fit some specified format, and one desires an encryption scheme mapping X to X. A classic algorithm due to Goldberg and Sipser (GS) [12] shows that, given a DFA for X, there exists an efficiently computable function $\mathsf{rank} : X \to \mathbb{Z}_N$, where $|X| = N$ and $\mathsf{rank}(x)$ is defined to be its position (its "rank") in the shortlex ordering of X. In addition, rank has an efficiently computable inverse $\mathsf{unrank} : \mathbb{Z}_N \to X$, so that $\mathsf{unrank}_L(i)$ is the i-th string in the ordering of L. Then to encrypt a string $x \in X$: (1) rank the input x to yield a number $a \leftarrow \mathsf{rank}(x)$, (2) encipher a, giving a new number b, then (3) unrank b to yield the ciphertext $y \leftarrow \mathsf{unrank}(b)$, which is an element of X.

BRRS focus on FPE for sets X that are a *slice* of a language L, that is $X = L \cap \Sigma^n$ for some n and where Σ is the alphabet of L. Relatedly, we define a *range-slice* of a language L as $X = L \cap (\Sigma^n \cup \Sigma^{n+1} \cup \cdots \cup \Sigma^m)$, for $n \leq m$. The latter is superior because it offers greater flexibility, although not explored by BRRS. Still, extending BRRS to an FPE scheme over the entire (regular) language is possible, by establishing a total ranking one slice at a time. The main disadvantage of the BRRS scheme is that it requires a DFA to represent the set X. For most users, this is an unnatural way to specify languages, or slices thereof.

We quickly note that the BRRS algorithm may be susceptible to timing-based side-channel attacks, since rank is not constant time. Timing information may therefore leak partial information about plaintexts. We leave to future work exploration of this potential security issue, which extends to libfte and other non-constant-time message encodings as well.

The FFX scheme. Bellare, Rogaway, and Spies [7] specify the FFX mode of operation, which is a specific kind of FPE scheme and is based on the BRRS work [5]. FFX takes a parameter $2 \leq r \leq 2^{16}$, the radix, and encrypts a plaintext $P \in L = \bigcup_\ell \{0, 1, \ldots, r-1\}^\ell$ to a ciphertext in L with $|L| = |P|$. The length ℓ ranges between a minimum value of 2 (or 10, if $r \geq 10$) and $2^{32} - 1$. For example, FFX[10] enciphers strings of decimal digits to (same length) strings of decimal digits; FFX[8] does likewise for octal strings. In addition, FFX has an extra "tweak" input, making it a length-preserving tweakable cipher, in the sense of [17]. The tweak allows FFX to support associated data.

We are aware of no public, open-source implementations of FFX, though there do exist proprietary ones [3]. Even given such an implementation, the formats supported by FFX are not as general as we might like. For example, the scheme does not support domain $\mathbb{Z}_N$ when N is not expressible as r^ℓ for the supported radices r. One can rectify this using cycle walking [8] but the burden is on developers to properly do so, hindering usability. Moreover, the user is left to determine how best to map more general formats into the set of formats that FFX supports.

Format-transforming encryption. Dyer, Coull, Ristenpart and Shrimpton (DCRS) [11] introduced the notion of format-*transforming* encryption, and gave a purpose-built scheme that mapped bitstring plaintexts to ciphertexts belonging to a specified regular language. Their FTE scheme was built to force protocol misidentification in state of the art deep-packet-inspection (DPI) systems used for Internet censorship.

The DCRS scheme is randomized, which lets it target strong privacy goals for the plaintexts (namely, semantic security [13]), and also naturally aligns with using standard encryption schemes as building blocks. The scheme itself is similar in spirit to BRRS: the plaintext bitstring is encrypted using an authenticated encryption scheme, the resulting intermediate ciphertext interpreted as a number, and this number is then unranked into the target language. Like BRRS, this scheme works on slices of a given regular language.

DCRS observe that regular expressions provide a friendlier programming interface for specifying inputs. But to use the GS scheme for ranking/unranking, they must first convert the given regular expression to an NFA and then from an NFA to a DFA. The last step often leads to a large blowup in the number of states, sometimes rendering the process completely intractable. (Examples of such regexs, and the associated NFA and DFA sizes, are given in Table 6 in Section 6.) Even when the process is tractable, the precomputed tables that DCRS and BRRS use to implement ranking require space that scales linearly in the number of states in the DFA. Many of the formats used by DCRS require several megabytes of memory; in one case, 383 MB. This is prohibitive for many applications, especially if one wants to keep several potential formats in memory.

Thus, in many instances it would be preferable to use the NFA representation of the given regex, but BRRS showed that ranking given just the NFA representation of a regular language is PSPACE-hard. Building any FPE or FTE scheme that works directly from an NFA has remained an open problem.

We also note that developers might hope for a general purpose FTE scheme, that takes arbitrary regular expressions for the input and output formats, and that can be built from existing deterministic cryptographic primitives (e.g., wideblock tweakable blockciphers) or randomized ones (e.g., authenticated encryption schemes). But actually instantiating such a scheme presents an array of algorithmic and engineering choices; in the current state of affairs, expert knowledge is required.

Summary. While a number of approaches to FPE and FTE exist, there is a gap between theory and developer-friendly tools. Implementations are non-existent, and even expert developers encounter challenges when implementing schemes from the literature, including: understanding and managing memory requirements, developing a "good" construction, or engineering the plaintext/ciphertext format pair. Finally, there exist fundamental performance roadblocks when using some classes of regular expressions. This is compounded by the fact that, a priori, it isn't obvious when a given regex will raise these roadblocks.

3 Overview of libfte

To aid adoption and usage of FPE and FTE, we developed a set of tools known collectively as libfte. At a high level, libfte has two primary components (see Figure 3): a standalone tool called the *configuration assistant*, and a library of algorithms (implemented in a mixture of Python and C/C++) that exposes an API for encryption and decryption via a number of underlying FPE/FTE schemes. Loosely, the API takes a configuration, describing what algorithms to use, and some key inputs for those algorithms, while the assistant helps developers determine good configurations. Let us start by talking about the assistant.

Configuration assistant. A *format* is a tuple $\mathcal{F} = (R, \alpha, \beta)$, where R is a regular expression, and $\alpha \leq \beta$ are numbers. A format defines a set of strings $L(\mathcal{F}) = \{\ s \in L(R) \ \big|\ \alpha \leq |s| \leq \beta\ \}$, where $L(R)$ is the set of strings matched by R. Following traditional naming conventions, we call $L(\mathcal{F})$ the language of the format. Because of its wide-spread use, in libfte the input R is specified in Perl-Compatible Regular Expression syntax. However, we note that PCRE syntax allows expressions that have no equivalent, formal regular expression. For instance, PCRE expressions using $\backslash 1, \backslash 2, ...$ (where $\backslash 1$ is a back-reference to the first capture group; see [1]) are not even context free, let alone regular. Thus, libfte accepts expressions built from a subset of the full PCRE syntax.

Our configuration assistant takes as input two formats, one describing the format of plaintext strings ($\mathcal{F}_P$), and one describing the desired format of ciphertext strings ($\mathcal{F}_C$). It also accepts some "preference parameters", for example specifying the maximum memory footprint of any scheme considered by the assistant, but these are set to some reasonable default value if not specified. It then runs a battery of tests, in an effort to determine which configurations will result in FPE/FTE scheme that abide by the user's inputs. Concretely, the assistant outputs a table listing various possible configurations (some configurations may not be possible, given the user's input), along with information pertaining to expected performance and memory usage. Given the user's preferences, the table lists the best option first. In the case that no available configuration is possible, the assistant provides information as to why, so that the user can alter their inputs and try again.

The encryption API. The algorithm library exposes an encryption API that takes as input an *encryption configuration*, which consist of a plaintext format, a ciphertext format, and a configuration identifier. The latter is a string that specifies the desired methods for performing ranking, unranking, encryption and decryption. The library performs all necessary precomputations (initialize rankers, build look-up tables, etc.) in an initialization function and returns a handle to an object that can perform encryption and decryption, according to the specified configuration. Currently, ten configurations are supported by libfte (see Section 6 for descriptions).

Roadmap. In Sections 4 and 5 we describe in detail the algorithms that result in these configurations. In Section 4 we detail a new type of ranking algorithm, what

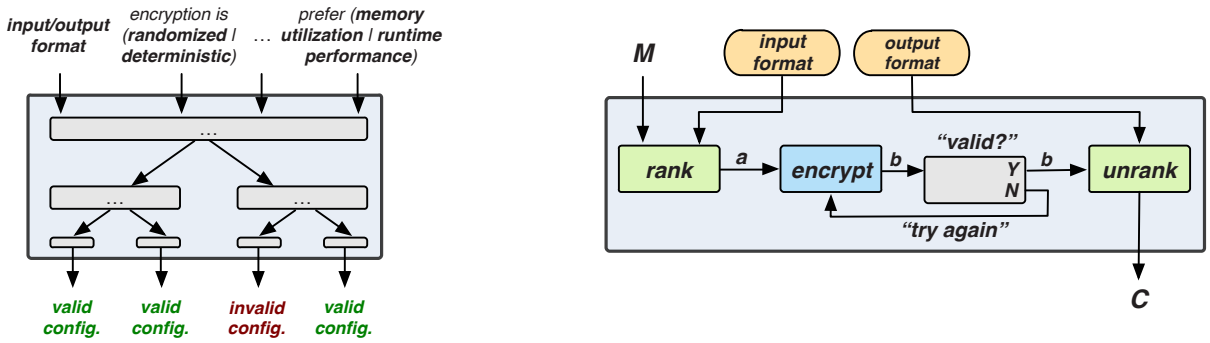


Figure 3: **Left:** The libfte configuration assistant (built against the library) helps users create formats that meet their specific performance requirements. The assistant takes an input/output format pair and uses a decision-tree process to determine if the formats are valid. If the formats are deemed valid, performance statistics are reported for the instantiated scheme(s). **Right:** The library implements APIs for FPE/FTE schemes. Shown is a diagram of the basic flow of our FPE/FTE schemes. As input it takes an input/output format and message M and returns a ciphertext C.

we call *relaxed ranking*, that allows us to work more directly with regular expressions (in particular, their equivalent NFAs), and sidestep the PSPACE-hardness obstacle. In Section 5, we lay out methods of combining relaxed ranking with standard cryptographic primitives to build both deterministic and randomized FPE and FTE schemes. For deterministic schemes, we leverage a technique called *cycle walking*, and for randomized schemes, we employ *rejection sampling*.

Then in Section 6 we describe specific instantiations of these schemes, and explain how the configuration assistant works in more detail. Finally, in Section 7 we show how these schemes can be put to work in three different use cases: database encryption, web form encryption, and network monitor circumvention.

4 Fast, Relaxed Ranking

The rank-encipher-unrank method for constructing FPE/FTE schemes needs efficient techniques for mapping strings in a regular language L to positive integers as well as computing the inverse operation (mapping positive integers back to strings in the language). Existing techniques are often impractical for two main reasons. First, the traditional DFA-based ranking requires the construction of a DFA corresponding to a regular expression. DFAs for some regular expressions can be very large. For instance, the minimum DFA for the regex $(a|b)^*a(a|b)\{20\}$ has $1 + 2^{21}$ states. Second, the numbers involved in ranking can be very large (for languages with many strings) and operations on these integers can therefore be computationally expensive. As an extreme example, ranking a $10,000$-byte long element accepted by the regex $.*$ requires numbers of up to $(2^8)^{10000}$ bits, or $10,000$ bytes. This section tackles these two challenges.

4.1 Relaxed Ranking

We introduce a framework for building FPE and FTE schemes directly from NFAs. The resulting algorithms will often use significantly less memory than the DFA approach, thus enabling general-purpose regex-based ranking in memory-constrained applications. For instance, the NFA for the regex $(a|b)^*a(a|b)\{20\}$ has 48 states.

A key insight is that we can circumvent the negative result about NFA ranking if we shift to a *relaxed ranking* approach, which we formally define in a moment. This will require, in turn, constructing FPE and FTE schemes given only relaxed ranking which we address in Section 5.

4.1.1 Relaxed Ranking Schemes

Informally, a relaxed ranking of a language $\mathcal{L}$ relaxes the requirement for a bijection from $\mathcal{L}$ to $\mathbb{Z}_{|\mathcal{L}|}$.

Formally, a *relaxed ranking scheme* for $\mathcal{L}$ is a pair of functions $\mathsf{Rank}_{\mathcal{L}}$ and $\mathsf{Unrank}_{\mathcal{L}}$, such that:

1. $\mathsf{Rank}_{\mathcal{L}} : \mathcal{L} \to \mathbb{Z}_i$ is injective, $i \geq |\mathcal{L}|$ (Note that we capitalize 'Rank' to distinguish relaxed ranking from ranking.)
2. $\mathsf{Unrank}_{\mathcal{L}} : \mathbb{Z}_i \to \mathcal{L}$ is surjective; and
3. For all $X \in \mathcal{L}$, $\mathsf{Unrank}_{\mathcal{L}}(\mathsf{Rank}_{\mathcal{L}}(X)) = X$.

The last condition means that we can correctly invert points in the image of $\mathcal{L}$, denoted $Img(\mathcal{L}) \subseteq \mathbb{Z}_i$. Note that a ranking is a relaxed ranking with $i = |\mathcal{L}|$.

DFA-based ranking revisited. As a thought experiment, one can view the traditional GS DFA-based ranking for regular languages as follows: let $\mathcal{I}$ be the set of all

accepting paths in a DFA. First, one maps a string $X \in \mathcal{L}$ to its accepting path $\pi_X \in \mathcal{I}$. Then, one maps π_X to an integer via an (exact) ranking. The composition of these two functions yields a ranking function for all strings in $\mathcal{L}$. In the DFA ranking algorithms of [5, 12], these two steps are merged.

A two-stage framework. We can use this two-step procedure to build efficient relaxed ranking algorithms. Suppose we desire to build a relaxed ranking function $\mathsf{Rank}_{\mathcal{L}}$ from a given set $\mathcal{L}$ into $\mathbb{Z}_i$. We first identify three components:

1. an *intermediate* set $\mathcal{I}$ for which we can efficiently perform ranking, i.e., there is an efficient algorithm for $\mathsf{rank}_{\mathcal{I}} : \mathcal{I} \to \mathbb{Z}_i$ where $i = |\mathcal{I}|$;
2. an injective function $\mathsf{map} : \mathcal{L} \to \mathcal{I}$; and
3. a surjective function $\mathsf{unmap}\colon \mathcal{I} \to \mathcal{L}$ such that for all $X \in \mathcal{L}$ it holds that $\mathsf{unmap}(\mathsf{map}(X)) = X$.

We then define

$$\begin{aligned} \mathsf{Rank}_{\mathcal{L}}(X) &= \mathsf{rank}_{\mathcal{I}}(\mathsf{map}(X)) \\ \mathsf{Unrank}_{\mathcal{L}}(Y) &= \mathsf{unmap}(\mathsf{unrank}_{\mathcal{I}}(Y)) \end{aligned}$$

Should unmap additionally be injective, then $\mathsf{Rank}_{\mathcal{L}}$ is a bijection, and we have (strict) ranking.

At first glance, this framework may seem to not have accomplished much as we rely on a strict ranking to realize it. But we will ensure that the language $\mathcal{I}$ allows for strict ranking, and so the framework allows us to transform the problem of ranking from a difficult set ($\mathcal{L}$) to an easier one ($\mathcal{I}$).

4.1.2 Relaxed Ranking Using NFAs

We construct relaxed ranking for NFAs using the approach above. We use as intermediate set $\mathcal{I}$ the set of all accepting paths in the NFA. To map into this set, for each string in $\mathcal{L}$ we deterministically pick an accepting path (a process called *parsing*). To rank on $\mathcal{I}$ we define a path ordering, and generalize the Goldberg-Sipser ranking algorithm for DFAs to count paths based on this ordering.

Recall that an NFA is a 5-tuple $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation[2], $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final (or accepting) states. If $(q, a, q') \in \delta$ then M may transition from state q to state q' when the current input symbol is a. We also write a transition $\tau = (q, a, q') \in \delta$ as $q \xrightarrow{a} q'$, where q is the source and q' is the destination of τ.

A *path* π in M is a sequence of transitions

$$q_{i_0} \xrightarrow{a_{j_1}} q_{i_1} \xrightarrow{a_{j_2}} q_{i_2} \cdots q_{i_{n-1}} \xrightarrow{a_{j_n}} q_{i_n} .$$

[2]We assume that there are no ϵ-transitions, but this is without loss of generality as there are standard methods to efficiently remove them from an NFA.

Path π can also be expressed as a sequence of transitions $\tau_1\tau_2 \cdots \tau_n$, where $n = |\pi|$ is the length of π. The suffix π^1 of the path π is $\tau_2 \cdots \tau_n$, and we have $\pi = \tau_1\pi^1$. The sequence of characters in the path is $\pi|_\Sigma = a_{j_1}a_{j_2}...a_{j_n}$.

The intermediate set $\mathcal{I}$. An accepting path is one that ends in an accepting state. Let $\mathrm{Acc}_{\mathrm{M}}(q)$ be the set of accepting paths starting from state q. We let $\mathcal{I} = \mathrm{Acc}_{\mathrm{M}}(q_0)$.

The functions map **and** unmap**.** We must map from $\mathcal{L}$ to $\mathcal{I}$ and back. The latter is simpler: define $\mathsf{unmap}(\pi)$ to be the word $\pi|_\Sigma$. This is fast to compute, in time linear in $|w|$. The forward direction $\mathsf{map}(w)$ requires a deterministic choice for an accepting path for w. This is called *parsing*. Any suitable parsing algorithm will work, but we note that the most obvious algorithms may be quite inefficient. For example, simply recording all accepting paths while running the NFA runs in time exponential in $|w|$ in the worst case.

Linear-time parsing. We now give the (to the best of our knowledge) first algorithm for determining a compact representation of all of an NFA's accepting paths for a string w. Then $\mathsf{map}(w)$ simply runs this algorithm for w and outputs the lexicographically least accepting path. Our algorithm constructs an implicit representation of a directed-acyclic graph (DAG) representing all accepting paths for w. The lexicographically least accepting path for w can then be found using a simple traversal of the DAG. Next we describe the algorithm in detail.

Let $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$ be an NFA, $Q' \subseteq Q$, and $c \in \Sigma$. We denote by $\delta(Q', c)$ the set of states q such that $(q', c, q) \in \delta$ for some $q' \in Q'$, and by $\delta^{-1}(Q', c)$ the set of states q such that $(q, c, q') \in \delta$ for some $q' \in Q'$.

Consider a string $w = c_1c_2...c_n$. Traditional NFA matching starts with a frontier of states $\mathcal{F}_0 = \{q_0\}$, and at every position k in w it computes $\mathcal{F}_k = \delta(\mathcal{F}_{k-1}, c_k)$. The string is accepted if $\mathcal{F}_n \cap F \neq \emptyset$. However, this does not allow easy recovery of an accepting path, even if all $\mathcal{F}_k$ sets are saved. The main reason for this is that there might be states in the frontiers that do not lead to an accepting state. To work around this, we also scan the input backwards, maintaining a backwards frontier set of states where $\mathcal{B}_n = F$, and $\mathcal{B}_{k-1} = \delta^{-1}(\mathcal{B}_k, c_k)$. Given the sequences $\{\mathcal{F}_k\}$ and $\{\mathcal{B}_k\}$, with $k = 0, ..., n$, we compute $\{\mathcal{S}_k\}$ where $\mathcal{S}_k = \mathcal{F}_k \cap \mathcal{B}_k$. The set $\mathcal{S}_k$ contains all states reachable from the start state following transitions on $c_1...c_k$ such that $c_{k+1}c_{k+2}...c_n$ is an accepting path. Together, $\{\mathcal{S}_k\}$ and the NFA transitions of the form (q, c_k, q') with $q \in \mathcal{S}_{k-1} \wedge q' \in \mathcal{S}_k$, form an implicit Direct Acyclic Graph (DAG) representation of all accepting paths for w. Finally, we traverse this DAG starting from $q_0 \in \mathcal{S}_0$ and following the lexicographically smallest transitions, which yields $\mathsf{map}(w)$.

NFA path ranking. All that remains is to give a strict

ranking algorithm for $\mathcal{I}$, the set of accepting paths in the NFA. Here, we can adapt techniques from the DFA-based ranking by Goldberg and Sipser. Their algorithm can be viewed as a recursive procedure for counting the number of accepting DFA paths that precede a given path in lexicographical order.

Let $T(q, n)$ be the number of paths of length n in $\mathrm{Acc}_{\mathrm{M}}(q)$. Note that, for all $q \in Q$ and $0 \leq i \leq n$, the value of $T(q, i)$ can be computed in polynomial time using a simple dynamic-programming algorithm.

Assume that the NFA transitions are enumerated according to a total ordering, and that $\tau \lessdot \tau'$ means that τ precedes τ' according to this order. The ordering on transitions induces a lexicographical ordering '$\prec$' on paths (which are sequences of transitions). Formally, if $\pi_1 = \tau_1\pi_1^1$ and $\pi_2 = \tau_2\pi_2^1$, then this order is:

$$\pi_1 \prec \pi_2 \iff \tau_1 \lessdot \tau_2 \vee \left(\tau_1 = \tau_2 \wedge \pi_1^1 \prec \pi_2^1\right) \quad (1)$$

Let $\mathsf{rank}(\pi)$ be the number of accepting paths $\pi' \prec \pi$ that precede π in the lexicographical order on paths. It follows that, $\mathsf{rank}(\epsilon) = 0$ (the rank of the empty string is 0), and for any $\pi = \tau\pi^1 \in \mathrm{Acc}_{\mathrm{M}}(q)$, we have:

$$\mathsf{rank}(\pi) = \mathsf{rank}(\pi^1) + \sum_{(q,c',q') \lessdot \tau} T(q', n-1) \quad (2)$$

Note that the sum is over transitions $\tau' = (q, c', q') \in \delta$ that precede τ in transition order, $\tau' \lessdot \tau$. In words, we are summing over all outgoing edges from q that lead to paths that are lexicographically smaller than the paths that follow the transition τ. Unrolling the recursion gives us an iterative procedure for ranking accepting paths of length n that can be efficiently implemented via dynamic programming.

To conclude, the relaxed ranking for a string w accepted by an NFA is $\mathsf{Rank}(w) = \mathsf{rank}(\mathsf{map}(w))$, and the reverse is $\mathsf{Unrank}(r) = \mathsf{unmap}(\mathsf{unrank}(r))$.

4.2 Large Integer DFA/NFA Optimization

We present a simple but effective optimization that speeds up both NFA and DFA-based ranking. In practice, ranking efficiency depends on how fast we evaluate the sum in equation (2), and this depends on the precise definition of the transition order. We define this order so that we can replace multiple large-integer additions with a single multiplication. Our experiments confirmed that this replacement indeed resulted in faster code.

Observe that equations (1) and (2) used for path ranking depend only on transition (edge) order and structure of the automaton. This observation is valid for both NFA and DFA. Previous, traditional, DFA ranking is given by these equations and standard lexicographical ordering, using character order: $(q, c', q') \lessdot (q, c'', q'') \Longleftrightarrow (c' < c'')$. In a DFA $c' = c'' \implies q' = q''$. But equation (1) does not have to use standard lexicographical ordering.

Our idea is to give *priority to states over characters.* We assume a state and character order given by an arbitrary but fixed enumeration of Q and Σ, and use the following order for transitions originating from the same state q: $(q, c', q') \lessdot (q, c'', q'')$ if-and-only-if $(q' < q'')$ or $q' = q''$ and $c' < c''$. This specific order allows for precomputation in equation (2). In equation (2) we can replace all the terms $T(q', n-1)$ which correspond to transitions $(q, c', q') \lessdot \tau$ with $n[q, q'] \times T(q', n-1)$, where the precomputed value $n[q, q']$ represents the number of transitions from q to q'. Similarly, all the terms corresponding to edges $\tau' = (q, c', q'')$, where $\tau' \lessdot \tau = (q, c'', q'')$, can be replaced by $r[q, c'', q''] \times T(q'', n-1)$, where $r[q, c'', q'']$ is the number of such transitions. These optimizations have benefit, because the numbers $T(q, n)$ can be very large multiple precision integers.

5 Building FTE Schemes

Now we turn to building FTE schemes, treating FPE in passing as a special case of FTE. We specifically give two methods for composing relaxed-ranking algorithms with an underlying cryptographic primitive to make an FTE scheme. For deterministic FTE, the cryptographic component is a tweakable cipher (e.g. FFX), and we call the composition *cycle-walking FTE*. For randomized FTE, the cryptographic component is an authenticated encryption scheme, and we call the composition method *rejection-sampling FTE.* (Impatient readers can look ahead to Figure 4 for the pseudocode.) We delay specific instantiations of the schemes until Section 6.1.

Informal FTE scheme syntax. We provide a formal treatment of FTE scheme syntax in the full version. We provide a simpler, more informal discussion of it here; this will suffice for what follows. An FTE scheme is a pair of algorithms $(\mathsf{Enc}, \mathsf{Dec})$. The FTE encryption algorithm Enc takes as inputs

- a key K
- a pair of formats $(\mathcal{F}_P, \mathcal{F}_C)$ that describe the language $L(\mathcal{F}_P)$ of plaintext inputs, and the language $L(\mathcal{F}_C)$ of ciphertext outputs
- a plaintext string $M \in L(\mathcal{F}_P)$
- associated data, and encryption parameters (both optional)

and outputs a ciphertext string $C \in L(\mathcal{F}_C)$, or a special "failure" symbol $\perp$. Associated data is data that must be bound to the underlying plaintext, but whose privacy is not required. (For example, metadata meant to provide context for the use or provenance of the plaintext.) We allow for encryption parameters to help enforce spe-

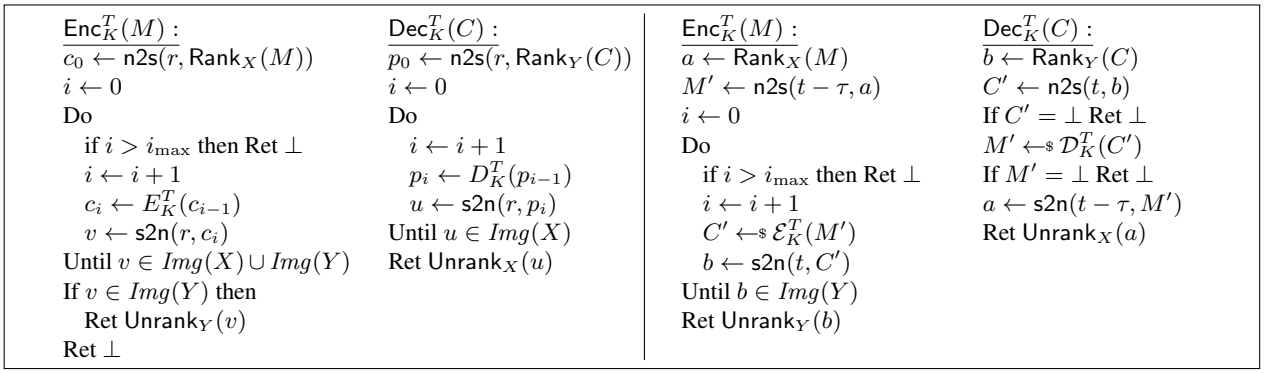


Figure 4: **Left:** Cycle-walking deterministic FTE. $\mathsf{n2s}(r, a)$ returns the string representing number a in radix r, and $\mathsf{s2n}(r, b)$ returns the number whose radix r representation is b. The parameter $i_{\max}$ determines the maximum number of iterations. **Right:** Rejection-sampling randomized FTE.

cific failure criteria, which will become clear when we describe our schemes. We write $\mathsf{Enc}_K^{T,P}(M)$ for FTE encryption of message M, under key K, using associated data T and parameters P. To ease the burden of notation (slightly), we typically do not explicitly list the parameters as inputs. The encryption algorithm may be randomized, meaning that fresh randomness is used for each encryption.

The FTE decryption algorithm Dec takes as input $(\mathcal{F}_P, \mathcal{F}_C), K$, a ciphertext C, and the associated data T (if any), and returns a plaintext M or $\perp$. The decryption algorithm is always deterministic.

Unlike conventional encryption schemes, we do not demand that $\mathsf{Enc}_K^{T,P}(M)$ always yield a valid ciphertext, or always yield $\perp$, when T, P and K are fixed. Instead, we allow encryption to "fail", with some small probability, to produce a ciphertext for a any given plaintext in its domain. Doing so will permit us to give simple and natural FTE schemes that would be ruled out otherwise.

In general, the formats can change during the lifetime of the key, even on a per-plaintext basis. (Of course, changes must be synchronized between parties.) When we talk about an FTE scheme being over some given formats, or their languages, we implicitly have in mind some notion of a format-session, during which the formats do not change.

5.1 Cycle-walking (deterministic) FTE

To build deterministic FTE schemes we take inspiration from BRRS rank-encrypt-unrank. However, accommodating format transformations and, especially, NFA-based language representations introduces new challenges.

To begin, let $X = L(\mathcal{F}_P)$ and $Y = L(\mathcal{F}_C)$. Assume that we perform relaxed ranking using the two-stage framework in Section 4.1.1, with the intermediate sets $\mathcal{I}(X)$ for X and $\mathcal{I}(Y)$ for Y. If Rank_X and Rank_Y are the corresponding relaxed-ranking functions, let $Img(X)$ be the image of X under Rank_X, and likewise $Img(Y)$ be the image of Y under Rank_Y. Define $N_X = |\mathcal{I}(X)|$ and $N_Y = |\mathcal{I}(Y)|$. (Recall that if we are using NFA-based ranking over either X or Y, these values can be significantly larger than $|X|$ or $|Y|$.) We assume that both N_X, N_Y are finite.

Say one has a tweakable cipher[3] E that natively supports strings over a variety of radices, e.g. FFX. (At a minimum, there are many constructions of secure tweakable ciphers that support radix 2, e.g. [9, 14, 15].) Now, fix integers $r \geq 2$ and $t \geq \lceil \max\{\log_r(N_X), \log_r(N_Y)\} \rceil$, so that a string of t symbols from $\{0, 1, \ldots, r-1\}$ suffices to represent the relaxed-rankings of X and Y. Then if E can encipher the set of strings $\{0, 1, \ldots, r-1\}^t$, we can encrypt a plaintext $M \in X$ as shown on the left side of Figure 4.

Cycle walking. A well-known fact about permutations is that they can be decomposed into a collection of disjoint cycles: starting from any element a in the domain of the permutation π, repeated application of π will result in a sequence of distinct values that eventually ends with a. Black and Rogaway [8] were the first to exploit this fact to build ciphers with non-standard domains, and we use it, too. For any fixed K and T, the mapping induced by E_K^T is a permutation. Thus, inside the Do-loop, the distinct strings $c_0, c_1 \leftarrow E_K^T(c_0)$, $c_2 \leftarrow E_K^T(E_K^T(c_0))$, and so on form a sequence that eventually must return to c_0. Intuitively, if we want a ciphertext that belongs to a particular subset $S \subseteq \{0, 1, \ldots, r-1\}^t$, we can walk the cycle until we hit a string $c_i \in S$.

There are, however, two important details to consider. The first is that encryption is not guaranteed to hit *any*

[3] If the FTE scheme does not need to support associated data, then the underlying cipher need not be tweakable, and references to T in the pseudocode can be dropped.

string $c_i \in S$. For example, if the subset is small, or the cycle is very short. So encryption must be equipped with test that tells it when this has happened, and $\perp$ should be returned. The second is that there must be a test that uniquely identifies the starting string c_0. This is because decryption should work by waking the cycle in reverse. Absent a test that uniquely identifies c_0, it may not be clear when the reverse cycle-walk should stop.

Our implementation deals with both of these issues. In particular, c_0 is the t-symbol string that results from relaxed-ranking our FTE plaintext input M. By definition, c_0 is a string that, when viewed as a radix-r integer, is in $Img(X)$. We desire to find a c_i that, when viewed as an integer, is in $Img(Y)$, since this is the set of integers that yield ciphertexts in Y that will be properly decrypted. Intuitively, the walk should halt on the first i for which this is true. But then, if any of $c_1, \ldots, c_{i-1}$ represent integers that are in $Img(X)$, proper decryption is not possible (because we do not know how many steps to go from c_i back to c_0). Thus our cycle-walking encryption checks, at each step, to see if the current walk should be terminated because decryption will not be possible, or because we have found a c_i that will yield a ciphertext Y that will decrypt properly. We also allow cycle-walking FTE to take a maximum-number-of-steps parameter $i_{\max}$, and encryption fails if that number of steps is exceeded.

Efficiency. The standard security assumption for a tweakable cipher is that, for any secret key K, and any associated data T, the mapping induced by E_K^T is indistinguishable from that of a random permutation. Modeling E_K^T as such, the expected number of steps before the cycle-walk terminates is at most $r^t/|Img(X) \cup Img(Y)|$ (a conservative bound) and never more than $i_{\max}$. Assuming the walk terminates before $i_{\max}$ steps, then the probability that the encryption succeeds is $p_s = |Img(Y)|/|Img(X) \cup Img(Y)|$. Since relaxed ranking is injective, $|Img(X)| = |X|$ and $|Img(Y)| = |Y|$, so $p_s \geq 1/(1 + |X|/|Y|)$. Thus we expect that p_s is quite close to 1 if $|Y| \gg |X|$.

Each step of the cycle-walk requires checking $v \in Img(X) \cup Img(Y)$, which can be done by checking $v \in Img(X)$ first (signaling termination of the walk), and then $v \in Img(Y)$ (signaling successful termination). A straightforward way to implement the last is to test if $v = \mathsf{Rank}_Y(\mathsf{Unrank}_Y(v))$ or, using our two-stage viewpoint on relaxed ranking, $\mathsf{map}(\mathsf{Unrank}(v)) = \mathsf{unrank}_{\mathcal{I}}(v)$, which may be faster. Checking $v \in Img(X)$ can be done likewise.

Recall that the NFA representation of a regex, unlike a DFA representation, may have many accepting paths for a given string in its language. This can lead to $N_X \gg |X| = Img(X)$ or $N_Y \gg |Y| = Img(Y)$, hence, potentially, $r^t \gg |Img(X) \cup Img(Y)|$. When this happens, the resulting in cycle-walking scheme may be prohibitively inefficient in some applications.

Simplifications. We note that the cycle-walking technique is used in [5], as well, but they restrict to the much simpler case that $X = Y$. More generally when we know that $Img(X) \subseteq Img(Y)$, we can simplify our construction. One may still need to cycle-walk in this case if $r^t > |Y|$. For example, say one desires to use $r = 2$ (binary strings) but the larger of $|X|, |Y|$ is not a power of two. But when $Img(X) \subseteq Img(Y)$ we know that, if the encryption cycle-walk terminates before $i_{\max}$ steps, then it always finds a point in $Img(Y)$, i.e. $p_s = 1$. Also, the expected number of steps is at most $r^t/|Img(Y)| = r^t/|Y|$, again modeling E_K^T as a random permutation. Finally, we note that the walk termination test can be simplified to $v \in Img(Y)$, and encryption can thereafter immediately return $\mathsf{Unrank}_Y(v)$.

Security. We mentioned, above, that the standard security assumption for a tweakable cipher is that, when the key K is secret, every associated data string T results in $E_K^T(\cdot)$ being indistinguishable from a random permutation. Under this assumption, it is not hard to see that the cycle-walking construction outputs (essentially) random elements of the set $Y = L(\mathcal{F}_c)$, when it does not output $\perp$. Intuitively, each $E_K^T(c_{i-1})$ in the cycle-walk is a random string (subject to permutivity), so the corresponding number v represented by the string is random, too. Thus, if $v \in Img(Y)$, it is a random element of this set, resulting in a random element of Y being chosen when v is unranked.

In the full version we formally define a security notion for deterministic FTE schemes, and give a theorem stating the security of our construction relative to this security notion.

5.2 Rejection-Sampling (randomized) FTE

We now turn our attention to building randomized FTE schemes. Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a conventional, randomized, authenticated-encryption scheme with support for associated data (AEAD). We assume that this scheme has a fixed ciphertext stretch τ; this is typical of in-use AEAD schemes. To build a randomized FTE scheme using a generalized ranking scheme, we use a rejection-sampling approach. Let t be the least integer such that both of the following are true: (1) $|\mathcal{I}(X)| \leq 2^{t-\tau}$, and (2) $|\mathcal{I}(Y)| \leq 2^t$. Then to encrypt $M \in X$, or decrypt $C \in Y$, under key K and associated data T, we do as shown on the right side of Figure 4.

A standard security assumption for AEAD schemes is that its ciphertexts are indistinguishable from strings (of the same length) that are uniformly random. Under this assumption, treating each C' as a random t-bit string, the

Sub-Component	Written in...	Lines of Code
Regular Expression Parser	C/C++/Flex/Bison	2,057
DFA Minimizer	C/C++	1,166
NFA/DFA Ranking	C/C++	2,752
FFX	C++	842
FPE/FTE	C++	870
Configuration Assistant	C++/Python	731

Table 5: The sub-components of libfte.

expected number of invocations of $\mathcal{E}_K^T$ is $2^t/|Img(Y)| = 2^t/|Y|$. (And certainly no more than $i_{\max}$.)

Under this standard security assumption, it is intuitive that any element of $Y = \mathcal{F}_c$ returned by our rejection-sampling FTE is a uniform one. If each C' is indistinguishable from a random string, then the corresponding number b represented by C' is random, too. Hence if $b \in Img(Y)$, then it is a random element of that set, and so the element of Y that results from unranking b will be random.

We give a formal security notion for randomized FTE, and a security theorem for rejection-sampling-based FTE, in the full version.

6 Realizing LibFTE

In Section 5 we explored strategies for constructing FPE/FTE schemes in theory. Now, let's concretely describe the schemes implemented in libfte.

Implementation. The libfte implementation is a hybrid of C, C++, Python, Flex and Bison. We present a detailed breakdown of the sub-components in Table 5. We engineered a custom regular expression parser because off-the-shelf solutions did not expose the appropriate data structures necessary to implement our NFA relaxed-ranking algorithm.

In addition to a native C++ interface, we also provide interfaces in Python and JavaScript for libfte. The Python interface is exposed through a manually-written wrapper around the C++ implementation. The JavaScript interface is provided through C++-to-JavaScript compilation.

6.1 Schemes Implemented in LibFTE

We use a shorthand notation to refer to types of libfte instantiations. As an example, T-ND-$ is a an F**T**E scheme that uses **N**FA-based ranking (Section 4.1) for the input format, and **D**FA-based ranking (Section 4.2) for the output format, and is randomized (**$**); T-ND denotes the same, but the scheme is deterministic. FPE constructions are similarly named, but begin with P.

For deterministic schemes (those without the final $) we use the cycle-walking construction, with FFX[2] as the underlying tweakable cipher. For randomized schemes, we use the rejection-sampling construction. As the underlying encryption scheme, we employ the Bellare-Rogaway "encode-then-encipher" paradigm [6], prepending the result of $\mathsf{Rank}_X(M)$ (interpreted as a fixed-length bitstring) with the appropriate number of random padding bits, and applying FFX[2] to this. Because our particular application of randomized FTE does not need support for associated data, the FFX tweak was fixed to an all-zeros string, and we do not need redundancy padding in our encode-then-encipher scheme.

We note that, although we fixed specific instantiations of FPE/FTE schemes for the sake of a concrete evaluation, there is no reason to restrict to these. In the randomized scheme, for example, one could use CTR-AES (with a random IV) and HMAC-SHA256 in an "encrypt-then-mac" composition [4, 18] (including any associated data in the mac-scope) for the underlying primitive.[4]

6.2 The LibFTE Configuration Assistant

We now turn our attention towards the implementation details of the libfte *configuration assistant*. We divide the internal workflow of the configuration assistant into three steps. First, we gather requirements from the user, this is done by the user passing parameters to a command-line interface. Then, we start with an initial set of all possible FPE/FTE schemes (i.e., P-xx, T-xx, T-xx-$) that one could instantiate, and use a decision tree algorithm to eliminate schemes from the initial set that do not satisfy user requirements. Finally, the configuration assistant analyzes the set of all schemes that were not eliminated in stage two, performs a battery of tests on them, and returns the results to the user. We provide a sample output of this tool in Figure 7.

Collecting requirements from the user. The command-line configuration assistant (see Figure 7) takes two required parameters, the input and output formats. In addition to the required parameters, the configuration assistant takes optional parameters, most notably: the memory threshold for the configuration assistant to determinize regexs, and the memory threshold for FPE/FTE scheme instantiation.

Identifying feasible schemes. Next, the configuration assistant's job is to eliminate schemes that fall outside the user-specified requirements. It starts with a set of all possible FPE/FTE schemes that one could construct (i.e.,

[4] One should keep in mind the interaction between the ciphertext length overheads of AEAD and the expected number of steps in rejection-sampling.

Scheme	Input/Output Format			DFA/NFA States	Memory Required	Encrypt (ms)	Decrypt (ms)
	R	α	β				
P-DD	`(a|b)*`	0	32	2	4KB	0.18	0.18
	`(a|b)*a(a|b){16}`	16	32	131,073	266MB	0.25	0.21
	`(a|a|b){16}(a|b)*`	16	32	18	36KB	0.19	0.18
	`(a|b){1024}`	1,024	1,024	1,026	34MB	1.2	1.2
P-NN	`(a|b)*`	0	32	3	6KB	0.36	0.35
	`(a|b)*a(a|b){16}`	16	32	36	73KB	0.61	0.60
	`(a|a|b){16}(a|b)*`	16	32	51	103KB	1,340	1,340
	`(a|b){1024}`	1,024	1,024	2,049	68MB	6.6	6.6

Table 6: Performance benchmarks for P-DD and P-NN constructions, based on our Python API. The regular expressions have been selected to highlight the strengths and weaknesses of the constructions. Recall that α and β are upper- and lowerbounds (respectively) on the length of strings used in a range slice.

```
$ ./configuration-assistant \
>  --input-format "(a|b)*a(a|b){16}" 0 32 \
>  --output-format "[0-9a-f]{16}" 0 16

==== Identifying valid schemes ====
WARNING: Memory threshold exceeded when
         building DFA for input format
VALID SCHEMES: T-ND, T-NN,
               T-ND-$, T-NN-$

==== Evaluating valid schemes  ====
SCHEME ENCRYPT DECRYPT ... MEMORY
T-ND   0.32ms  0.31ms  ... 77KB
T-NN   0.39ms  0.38ms  ... 79KB
...
```

Figure 7: A sample execution of our configuration assistant for building an FTE scheme. The tool failed to determinize the regex of the input format, and notifies the user that that T-ND, T-NN, T-ND-$ and T-NN-$ constructions are possible. Then reports on the performance of these schemes.

P-xx, T-xx, T-xx-$). If the DFA can't be built (within the user-specified memory thresholds) for the input format, then we eliminate P-Dx, T-Dx and T-Dx-$ schemes from consideration. We repeat this process for the output format. Then we perform a series of additional checks — involving the sizes of $L(\mathcal{F}_P)$, $L(\mathcal{F}_C)$, the sizes of the intermediate representations, the minimum ciphertext stretch of underlying cryptographic primitives, etc. — to cull away schemes that should not be considered further.

Evaluating feasible schemes. Finally, we consider the set of schemes that remain from the previous step. If there are none, we output an error. Otherwise, we iterate through the set of schemes and perform a series of functional and performance tests against them. We have fourteen quantitative tests, such as: the average time elapsed to perform encryption/decryption, the (estimated) probability that encryption returns $\perp$, and memory requirements. The final result of the tool is a table output to the user, each row of the table represents one scheme and the columns are results from the quantitative tests performed. The method for sorting (i.e., prefer memory utilization, prefer runtime performance, etc.) is a user-configurable parameter.

6.3 Performance

We conclude this section with benchmarks of our libfte implementation. All benchmarks were performed on Ubuntu 12.04, with an Intel Xeon E3-1220 at 3.10GHz and 32GB of memory. Numbers reported are an average over 1,000 trials for calls to our libfte Python API. For memory utilization of each scheme, we measure the memory required at runtime to use the specified formats. For encrypt benchmarks we select a random string in the language of the input format and measure the time it takes for encrypt to return a ciphertext. For decrypt benchmarks we select a random plaintext, encrypt it, then measure the time it takes for decrypt to recover the plaintext.

In Table 6 we have the performance of libfte for P-DD and P-NN schemes. Note that `(a|b)*a(a|b){16}` requires roughly four orders of magnitude less memory using a P-NN scheme, compared to a P-DD scheme. With the P-NN scheme for `(a|a|b){16}(a|b)*`, the high encrypt cost is completely dominated by cycle walking, we do roughly 720 FFX[2] encrypt calls per P-NN call. (The configuration assistant would inform the user of this, and the user would have the opportunity to re-write the expression as `(a|b){16}(a|b)*`.) For `(a|b)*`, the two constructions (i.e., P-DD, P-NN) require a comparable amount of memory but the P-DD construction is twice as fast for encryption/decryption.

Due to space constraints we omit benchmarks for T-xx and T-xx-$ schemes in this section, and defer their analysis to Section 7.

7 Exploring LibFTE Use Cases

We turn our attention to integrating libfte into third-party applications. Our goal is to show that libfte is easy to use, flexible, and in some cases *improves* performance, compared to other cryptographic solutions. In this section we consider three use cases: database encryption, web form encryption, and encryption using formats lifted from a network monitoring device.

7.1 Databases

We start with integration of libfte into a PostgreSQL database. For our database we used PostgreSQL 9.1 in its default configuration. Our server was Ubuntu 12.04 with an Intel Xeon E3-1220 v3 @ 3.10GHz and 32GB of memory. We use the pgcrypto library that is included in PostgreSQL's contrib directory as our baseline cryptographic library. We performed all tests with the PostgreSQL client and server on the same machine, such that network latency did not impact our performance results.

The integration of libfte into our database as PostgreSQL stored procedures required 53 lines of plpython code.

Pgbench is tool included with PostgreSQL to measure database performance. As input, pgbench takes a description of a database transaction and runs the transaction repeatedly, emulates concurrent users, and measures statistics such as transactions per second and response latency. We used pgbench's default database schema and transaction set for testing libfte's impact on PostgreSQL performance. The default pgbench testing schema simulates a simple bank and has four tables: accounts, tellers, branches, and history. The default transaction set for testing includes three query types: SELECT (S), UPDATE (U) and INSERT (I). There are three different transaction types that can be selected using pgbench: S, USI, and USUUI — for each transaction type the acronym represents the type and order of queries executed.

In order to test the performance impact libfte has on PostgreSQL, we created four configurations for populating and accessing data in the database:

- **PSQL:** The default configuration and schema used by the pgbench utility for its simple bank application. No encryption is employed.
- **+AES:** The default schema, with the following modifications: the balance columns in accounts, tellers, and branches are changed from type `integer` to type `bytea`. To secure these fields we use AES128 in ECB mode with PKCS padding.
- **+AE:** We use the same schema as **+AES**, but we change our encryption algorithm to pgcrypto's recommended encrypt function `pgp_sym_encrypt`, which provides privacy and integrity of data.
- **+FPE:** We use the default schema, but employ a libfte P-DD scheme with the format $R \leftarrow \backslash\text{-}[0\text{-}9]\{9\}$, $(R, 9, 10)$, to encrypt account balances (in accounts, tellers, and branches) in-place.

Account Balance Queries				
Transaction Type	**Transactions Per Second**			
	PSQL	+AES	+AE	+FPE
S	38,500	30,246	8,380	8,280
USI	2,380	2,259	1,580	1,540
USUUI	99.2	97.5	97.2	96.5

Table 8: A comparison of throughput (transactions/second) for our four database configurations.

We note that in our evaluation we did not have the option to compare libfte to a scheme that provides the same functionality or security, as no prior scheme exists. We compare to **+AES** because it provides a baseline performance that we would not expect libfte to exceed. The comparison to **+AE**, which provides privacy and integrity, can be used as a baseline for the performance of a cryptographic primitive implementation in a widely-used, mature database product.

Performance For each configuration and transaction type we executed pgbench for five minutes with 50 active customers, leaving all other pgbench parameters as default. In all configurations except PSQL, when performing modifications to the database we perform an encrypt when storing the account balance. When retrieving the account balance we recover the plaintext via a call to the decryption algorithm.

In Table 8 we have the have the benchmark results for transactions per second carried out by the server for our four database configurations and three transaction types. For the most complex transaction type (USUUI) our +FPE configuration reduces the number of transactions per seconds by only 0.8%, compared to the +AE configuration. For the simplest query type (S), +FPE reduces the transactions-per-second rate by only 1.1%, compared to +AE. Compared to the +AES configuration the +AE and +FPE reduce the transactions per second by roughly 72%. This is, in fact, not surprising as under the hood the +FPE configuration relies on FFX, which in turn calls AES at least ten times.

In Table 9 we have the average latency for each of the five different query types. This measures the amount of time elapsed between a client request and server response. Compared to the +AE configuration, +FPE introduces no substantial latency.

Simultaneous encryption and compression. As one final test, we deploy a T-DD scheme in our PostgreSQL

Account Balance Queries				
	Average Latency (ms)			
Query	PSQL	+AES	+AE	+FPE
(U) accounts	0.6	1.2	2.1	2.1
(S) accounts	0.4	0.5	1.0	1.0
(U) tellers	412	412	415	420
(U) branches	78	80	80	84
(I) history	0.2	0.2	0.2	0.2

Table 9: Average latency per query for each database configuration.

Credit Card Number Queries					
	PSQL	+AES	+AE	+FPECC	+FTECC
Table Size	50MB	65MB	112MB	50MB	**42MB**
Query Avg.	74ms	92ms	112ms	125ms	110ms

Table 10: FTE for simultaneous encryption and compression. The table size is the amount of space required on disk to store the table. We also present the average query time (over 1,000 trials) for selecting (and decrypting) 100 credit card numbers at random from 1 million records.

database to provide simultaneous privacy and compression. We augment the default pgbench database schema to add a new table for credit card numbers. This table has two columns: an account number of type `integer` and a credit card number field of type `bytea`. (Following the structure of the pgbench schema, we do not add any indexes to this table.) We start with the four configurations we presented in our initial benchmarks. However, we change our +FPE configuration to a P-DD scheme that encrypts 16-digit credit card number in-place and call this +FPECC. Then we introduce a new configuration, +FTECC, a T-DD scheme where our input format is a 16-digit credit card number and our output format is the set of all 7-byte strings.

In each configuration we populated our database with 1 million random credit card numbers. For each database configuration, we have a breakdown (Table 10) of the query cost to retrieve 100 credit card numbers at random (and decrypt, if required) as well as the total size of the new credit card table. Compared to the +AES and +AE configuration, our +FTECC configuration requires 35% and 62.5% less space, respectively. We note that it may be possible to employ additional compression in the PSQL, +FPECC settings (e.g., an int to bitstring conversion). However, such optimizations are not possible in the +AES and +AE configurations.

We also highlight that, with respect to query times (Table 10) our +FTECC configuration modestly outperforms the +AE configuration. Compared to +AES, +FTECC introduces a 19.5% increase in query cost.

7.2 Web Forms

Next, we present a Firefox extension powered by libfte. The job of this browser extension is to encrypt sensitive contact information, client-side, in a Yahoo address book contact form, prior to submission to the remote Yahoo servers.

Browser extension. We tested our libfte-extension with Firefox version 26, using our C++-to-JavaScript compiled libfte API. In addition to libfte , in roughly 200 lines of code, we implemented logic that was responsible for locating page elements to encrypt/decrypt. The Yahoo-specific logic was minimal and consisted of mappings between form fields and FPE/FTE schemes.

Fields were manually specified using unique identifiers (e.g., CSS id tags) and mapped to their corresponding P-DD FPE scheme in JavaScript. There are many options for determining what input/output formats to use for a given scheme. For this proof-of-concept we started with a set of formats we considered to be reasonable, then progressively relaxed/increased constraints on the formats appropriately until they were accepted by Yahoo's server-side validation. As a couple examples, we found that the email address field is required to have an @ symbol, and all dates are required to be valid date ranges. (e.g., month must between 1 and 12 inclusive) We expect that such constraints could be identified programmatically, at scale, using a browser automation tool such as Selenium [2].

Our Firefox extension exposes an "encrypt/decrypt" drop-down menu to the user. Prior to saving a new contact to their address book, the user can press the "encrypt" button to automatically libfte-encrypt all fields in the form. To recover the encrypted data, they user visits the page for a contact, then presses the libfte extension's decrypt button to recover the plaintext data. With further engineering efforts this encryption/decryption process could be transparent to the user. We present a screenshot of our extension in Figure 11.

7.3 Network Monitors

Finally, we turn our attention to building T-xx-$ schemes (as used in [11]) by lifting regular expressions from the Snort IDS [19]. As far as these authors are aware, this Snort IDS corpus of regular expressions is the largest and most diverse (publicly available) set of regexes used for deep-packet inspection.

In the initial exploration of FTE by Dyer et al. [11] a fundamental limitation to their construction was the need to perform a regex-to-DFA conversion for formats. Unfortunately, this creates a natural asymmetry: systems

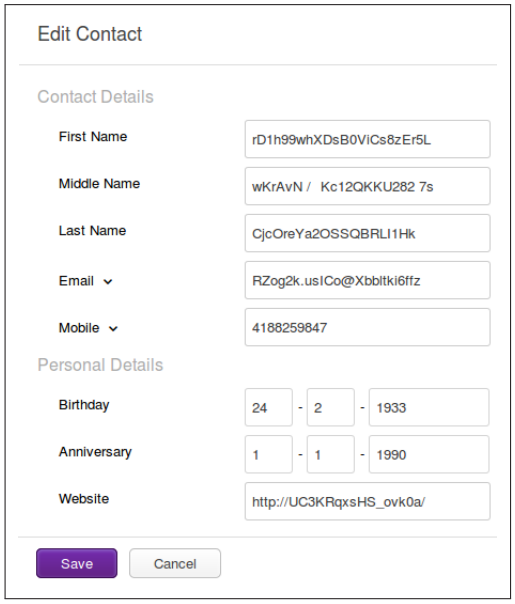


Figure 11: Screenshot from our Firefox Browser Extension that encrypts the data fields of our Yahoo address book, client-side, prior to transmission to the Yahoo servers.

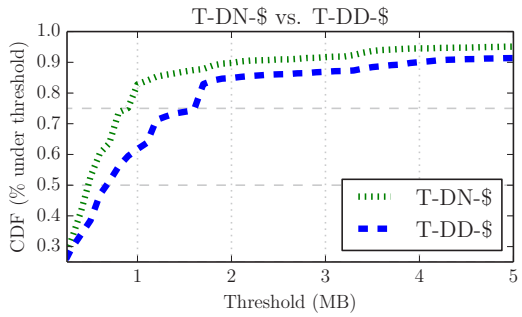


Figure 12: The CDF representing the fraction of the Snort corpus that can be instantiated for a given memory threshold. The CDF graph has a long tail and reaches 100% at 143MB for T-DN-$. We were unable to calculate the threshold for T-DD-$ to reach 100%, due to memory constraints on our test system.

such as Snort are able to perform network monitoring directly from an NFA representation, but the construction presented by Dyer et al. requires regex-to-DFA conversion. In this section we show that we've overcome this limitation — regular expressions that were intractable using the construction by Dyer et al. are no longer an obstacle, given our new NFA ranking algorithm.

Snort IDS regex corpus. To build our corpus, we started with the official Snort ruleset, version 2955, released Jan 14, 2014. Each rule in the ruleset contains a mixture of values, including static strings or regular expressions to match against traffic. From each rule we extracted all regular expressions (as defined by the `pcre` field) which resulted in 6,277 expressions. Of these, 3,458 regular expressions compiled with our regular expression parser[5]. For all regular expressions that compiled, if we were able to instantiate their precomputation table in memory, we were able to successfully utilize them for encryption.

Corpus evaluation. For each regular expression R in the Snort corpus we attempted to build a T-DD-$ and T-DN-$ scheme with an output format $\mathcal{F} \leftarrow (R, 0, 256)$, and input format that is a $\lfloor \log_2 |L(\mathcal{F})| \rfloor$-bit string. This choice of libfte scheme and α and β is motivated by the construction in [11].

In Figure 12 we plot the CDF of the fraction of the corpus that can be instantiated when constrained by a given memory threshold, for each scheme. At 1MB, roughly 60% of the corpus can be instantiated using T-DD-$ ranking, compared to roughly 85% of the corpus with T-DN-$ ranking. At 5MB, T-DN-$ is at roughly 97% and T-DD-$ is at roughly 92%. If we increase the threshold to 143MB (beyond the focus of the graph) we can instantiate 100% of the corpus using T-DN-$. Yet, at at threshold of 1GB, we are able to instantiate only 97.0% of the corpus using T-DD-$. In fact we were unable to construct some schemes (due to memory constraints) using T-DD-$, so we don't know the exact threshold required to reach 100% instantiation.

As a final test we measured the *total* memory utilization for instantiating the complete Snort corpus. Initially, we instantiated all regular expressions in the corpus using T-DD-$, which required a cumulative 8.8GB of memory. Then we considered a "best case" scenario, where, over the 97% of tractable regexs (those that we could construct a T-DD-$ scheme) we took the minimum of the T-DD-$ or T-DN-$ memory utilization. Using the best-case approach we reduced memory utilization to 6.2GB, a reduction of roughly 30%. The best-case scenario is, of course, biased against T-DN-$, as 3% of the corpus couldn't be instantiated with T-DD-$.

8 Conclusion

In this paper we presented a unifying approach for deploying format-preserving encryption (FPE) and format-transforming encryption (FTE) schemes. The approach is realized via a library we call libfte, which has two components: an offline configuration assistant to aid engineers in developing formats and understanding their system-level implications, and an API for instantiating

[5]We don't support greedy operators `*?` or case insensitivity `(?i...)`, which accounted for the majority of compilation failures. Greedy operators are used for parsing, not for language definition, and we do not support extended patterns of the form `(?...)` in general.

and deploying FPE/FTE schemes. In the development of libfte we overcame a number of obstacles. Most notably, we developed a new approach to perform FPE/FTE directly from the NFA representation of a regular expression, which was previously considered to be impractical. This significantly increases the expressiveness of regular languages for which FTE can be made useful in practice, and generally improves system efficiency, potentially making FTE a viable cryptographic option in contexts where it previously was not. We studied libfte performance empirically in several application contexts, finding that it typically introduces negligible performance overhead relative to conventional encryption. In some cases (e.g. simultaneous compressions and encryption) even enables substantial performance improvements.

Our work surfaces many avenues for future research. To name a few: investigate the security of libfte's algorithms (and FTE implementations, in general) in the face of side-channel attacks; integrate FTE into additional applications, and report on newly found algorithmic and engineering challenges; and explore efficiency improvements for specific classes of regular expressions that are in wide use. To promote further research and development, we will make libfte open source and publicly available at `https://libfte.org/`.

References

[1] Perl regular expressions man page. http://perldoc.perl.org/perlre.html, February 2014.

[2] Seleniumhq: Browser automation. http://www.seleniumhq.org/, February 2014.

[3] Voltage security. http://www.voltage.com/, February 2014.

[4] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. pages 531–545, 2000.

[5] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, pages 295–312. Springer-Verlag, 2009.

[6] Mihir Bellare and Phillip Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. pages 317–330, 2000.

[7] Mihir Bellare, Phillip Rogaway, and Terence Spies. The ffx mode of operation for format-preserving encryption draft 1.1, 2010.

[8] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In *Topics in Cryptology–CT-RSA 2002*, pages 114–130. Springer Berlin Heidelberg, 2002.

[9] Debrup Chakraborty and Mridul Nandi. An improved security bound for HCTR. pages 289–302, 2008.

[10] Debra L. Cook, Angelos D. Keromytis, and Moti Yung. Elastic block ciphers: the basic design. pages 350–352, 2007.

[11] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM Conference on Computer and Communications Secuirty (CCS 2013)*, November 2013.

[12] A Goldberg and M Sipser. Compression and ranking. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 440–448, New York, NY, USA, 1985. ACM.

[13] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[14] Shai Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. pages 315–327, 2004.

[15] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. pages 482–499, 2003.

[16] Daniel Lachaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. LibFTE: A toolkit for constructing practical, format-abiding encryption schemes (full version), 2014. Available from authors' websites.

[17] Moses Liskov, Ronald L Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology-CRYPTO 2002*, pages 31–46. Springer, 2002.

[18] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. Reconsidering generic composition. In *Advances in Cryptology – EUROCRYPT '14*, LNCS, pages 257–274. Springer-Verlag, 2014.

[19] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[20] Phillip Rogaway. A synopsis of format-preserving encryption. Unpublished manuscript, March 2010.

[21] Rich Schroeppel. An overview of the hasty pudding cipher, July 1998.

Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens

Daniel Demmler, Thomas Schneider, and Michael Zohner

Technische Universität Darmstadt, Germany
{daniel.demmler,thomas.schneider,michael.zohner}@ec-spride.de

Abstract

Secure two-party computation allows two mutually distrusting parties to jointly compute an arbitrary function on their private inputs without revealing anything but the result. An interesting target for deploying secure computation protocols are mobile devices as they contain a lot of sensitive user data. However, their resource restriction makes the deployment of secure computation protocols a challenging task.

In this work, we optimize and implement the secure computation protocol by Goldreich-Micali-Wigderson (GMW) on mobile phones. To increase performance, we extend the protocol by a trusted hardware token (i.e., a smartcard). The trusted hardware token allows to pre-compute most of the workload in an initialization phase, which is executed locally on one device and can be pre-computed independently of the later communication partner. We develop and analyze a proof-of-concept implementation of generic secure two-party computation on Android smart phones making use of a microSD smartcard. Our use cases include private set intersection for finding shared contacts and private scheduling of a meeting with location preferences. For private set intersection, our token-aided implementation on mobile phones is up to two orders of magnitude faster than previous generic secure two-party computation protocols on mobile phones and even as fast as previous work on desktop computers.

1 Introduction

Secure two-party computation allows two parties to process their sensitive data in such a way that its privacy is protected. In the late eighties, Yao's garbled circuits protocol [Yao86] and the protocol of Goldreich-Micali-Wigderson (GMW) [GMW87] showed the feasibility of secure computation. However, secure computation was considered to be mostly of theoretical interest until the Fairplay framework [MNPS04] demonstrated that it is indeed practical. Since then, many optimizations have been proposed and several frameworks have implemented Yao's garbled circuits protocol (e.g., FastGC [HEKM11]) and the GMW protocol (e.g., the framework of [CHK+12]) on desktop PCs.

Motivated by the advances of secure computation on desktop PCs, researchers have started to investigate whether secure computation can also be performed in the mobile domain. Mobile devices, in particular smartphones, are an excellent environment for secure computation, since they accompany users in their daily lives and typically hold contact information, calendars, and photos. Users also store sensitive data, such as passwords or banking information on their devices. Moreover, typical smartphones are equipped with a multitude of sensors that collect a lot of sensitive information about their users' contexts. Therefore, it is of special importance to protect the privacy of data handled in the mobile domain.

In contrast to desktop PCs, mobile devices are rather limited in computational power, available memory, communication capabilities, and most notably battery life. Although mobile phones have seen an increase in processing speed over the past years, they are still about one order of magnitude slower than typical desktop computers when evaluating cryptographic primitives (cf. §5.4). These differences are due to the CPU architectures having a more restrictive instruction set and being optimized for low power consumption rather than performance, since mobile devices are battery-powered and lack active cooling. Moreover, the limited size of the main memory requires the programmer to carefully handle data objects in order to avoid costly garbage collections on Java-based Android smartphones. Network connections of mobile devices are almost exclusively established via wireless connections that have lower bandwidth and higher, often varying latency compared to wired connections. Tasks that are computationally in-

tensive or require long send/receive operations should be avoided when a mobile device is running on battery, as such tasks quickly drain the battery charge and thereby reduce the phone's standby time. Instead, such operations could be pre-computed when the mobile device is connected to a power source, which usually happens overnight. These limitations pose a big challenge for efficient secure computation and cause generic secure computation protocols to be several hundred times slower on mobile devices than on desktop PCs [HCE11], even in the semi-honest adversary model.

To enable secure two-party computation in the mobile domain, solutions have been developed that outsource secure computation to the cloud, e.g., [KMR12, Hua12, CMTB13]. However, recent events have shown that cloud service providers can be forced to give away data to third parties that are not necessarily trusted, such as foreign government agencies. Even if the employed protocols ensure that the cloud provider learns no information about the users' sensitive data, he can still learn and hence be forced to reveal meta-information such as the frequency of access, communication partners involved, the computed function, or the size of the inputs. Moreover, these server-aided approaches require the mobile device to be connected to the Internet which might not be possible in every situation or may cause additional costs.

An alternative solution, which we also use in this work, is to outsource expensive operations to a trusted hardware token that has very limited computational resources and is locally held by one of the communication partners.[1] Such hardware tokens are increasingly being adopted in practice, e.g., trusted platform modules (TPMs). Their adoption is particularly noteworthy on mobile devices in the form of smartcards that are the basis for subscriber identity modules (SIM cards), as well as for mobile payment or ticketing systems. A first approach for outsourcing Yao's garbled circuits protocol to such a trusted hardware token was proposed in [JKSS10]. However, this protocol requires the function to be known in advance and uses costly symmetric cryptographic operations during the online phase. We give an alternative solution that removes these drawbacks.

1.1 Outline and Our Contributions

In this work, we introduce a scheme for token-aided ad-hoc generic secure two-party computation on mobile devices based on the GMW protocol. After introducing preliminaries (§2) we detail our setting and trust assumptions that are similar to the ones in a TPM scenario (§3). We outline how a trusted hardware token can be used to shift major parts of the workload into an initialization phase that can be pre-computed on the token, independently of the later communication partner (§4), e.g., while the mobile device is charging. We thereby obtain a token-aided scheme that is well-suited for efficient and decentralized (ad-hoc) secure computation in the mobile domain. We implement and evaluate our scheme (§5) and demonstrate its performance using typical secure computation applications for mobile devices, such as securely scheduling a meeting with location preferences and privacy-preserving set intersection (§6). We compare our scheme to related work (§7) and conclude and present directions for future work (§8). More detailed, our contributions are as follows.

Token-Aided Ad-Hoc Secure Two-Party Computation on Mobile Devices (§4) We develop a token-aided secure computation protocol which offloads the main workload of the GMW protocol to a pre-computation phase by introducing a secure hardware token $\mathcal{T}$, held by one party $\mathcal{A}$ (cf. §3). $\mathcal{T}$ is issued by a trusted third party and provides correlated randomness [Hua12, Chap. 6] to both parties that is later used in the secure computation protocol. To prepare the secure computation, the other party $\mathcal{B}$ obtains seeds for his part of the correlated randomness from $\mathcal{T}$ via an encrypted channel. To further increase flexibility, we describe how to make the pre-computation independent of the size of the evaluated function $|f|$, at the cost of a $t \cdot \log_2 |f|$ factor communication overhead between $\mathcal{T}$ and $\mathcal{B}$, where t is the symmetric security parameter. In contrast to Yao-based approaches [MNPS04, JKSS10, HCE11, HEK12] and previous realizations of the GMW protocol [CHK+12, SZ13, ALSZ13], our protocol offers several benefits as summarized in Tab. 1 (cf. §4.5 for details).

Table 1: Comparison with related work.

Property	Yao [HCE11]	Token Yao [JKSS10]	GMW [CHK+12]	Ours §4
f unknown in init phase	✓	✗	✓	✓
ad-hoc communication $\ll t \cdot \lvert f \rvert$	✗	✓	✗	✓
$\ll \lvert f \rvert$ crypto operations in ad-hoc phase	✗	✗	✗	✓

Implementation (§5) We implement our token-aided protocol for semi-honest participants and evaluate its performance using two consumer-grade Android smartphones and an off-the-shelf smartcard. Thereby, we provide an estimate for the achievable runtime of generic se-

[1] This locality is also a security feature, as external adversaries either need to corrupt the token before it is shipped to the user or later get physical access to break into it.

cure computation in the mobile domain. Our implementation enables a developer to specify the functionality as a Boolean circuit, which can, for instance, be generated from a high-level specification language. We show that the performance of our token-aided pre-computation phase is comparable to interactively generating the correlated randomness using oblivious transfer.

Applications (§6) We demonstrate the practical feasibility of the GMW protocol on mobile devices by performing secure two-party computation on two smartphones using various privacy-preserving applications such as availability scheduling (§6.1), location-aware scheduling (§6.2), and set-intersection (§6.3). Most notably, for private set-intersection, our token-aided scheme outperforms related work that evaluates generic secure computation schemes on mobile devices [HCE11] by up to two orders of magnitude and has a performance that is comparable with secure computation schemes that are executed in a desktop environment [HEK12].

2 Preliminaries

In the following, we define our notation (§2.1) and the ad-hoc scenario (§2.2), and give an overview of oblivious transfer (§2.3) and the GMW protocol (§2.4). We describe Yao's garbled circuits in the full version [DSZ14].

2.1 Notation

We denote the two parties that participate in the secure computation as $\mathcal{A}$ and $\mathcal{B}$. We use the standard notation for bitwise operations, i.e., $x \oplus y$ denotes bitwise XOR, $x \wedge y$ bitwise AND, and $x||y$ the concatenation of two bit strings x and y. We refer to the symmetric security parameter as t and the function to be evaluated as f.

2.2 Ad-Hoc Scenario

In an *ad-hoc* secure two-party computation scenario, two parties that do not necessarily know each other in advance want to spontaneously perform secure computation of an arbitrary function f on their private inputs x and y. Traditionally, secure computation protocols consist of two interactive phases: the *setup phase* (independent of x and y) and the *online phase*. We extend this setting by a local *init phase* as depicted in Fig. 1.

The init phase takes place at any time before the parties have identified each other and is used for preprocessing. In the setup phase, the parties have determined their communication partner, establish a communication channel, and know an upper bound on the function size $|f|$. In the online phase, the parties provide their private inputs x and y to the function f that they want to evaluate and begin the secure computation. The *ad-hoc time* is the combined time for setup and online phase.

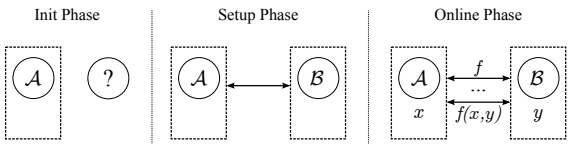


Figure 1: The three secure computation phases.

2.3 Oblivious Transfer

Oblivious transfer (OT) is a fundamental building block for secure computation. In an OT protocol [Rab81], the sender inputs two strings (s_0, s_1). The receiver inputs a bit $c \in \{0,1\}$ and obtains s_c as output without revealing to the sender which of the two messages he chose and without the receiver learning any information about s_{1-c}. OT protocols, such as [NP01], require public-key cryptography and make OT a relatively costly operation. OT extension [IKNP03] allows to increase the efficiency of OT by extending a small number of t base OTs to a large number $n \gg t$ of OTs whilst only using $\mathcal{O}(n)$ symmetric cryptographic operations. Optimizations to the OT extension protocol of [IKNP03] were suggested in [ALSZ13], which allow the parties to reduce the amount of data sent per OT. Moreover, [ALSZ13] describes a more efficient variant of the OT extension protocol for computing random OT, where the sender obtains two random values as output of the OT protocol.

2.4 The GMW Protocol

In the GMW protocol [GMW87], two (or more) parties compute a function f, represented as Boolean circuit on their private inputs by secret sharing their inputs using an XOR secret sharing scheme and evaluating f gate by gate. Each party can evaluate XOR gates locally by computing the XOR of the input shares. AND gates, on the other hand, require the parties to interact with each other by either evaluating an OT or by using a *multiplication triple* [Bea91] as shown in the full version [DSZ14]. Finally, all parties send the shares of the output wires to the party that shall obtain the function output. The main cost factors in GMW are the total number of AND gates in the circuit, called (multiplicative) size $|f|$, and the highest number of AND gates between any input wire and any output wire, called (multiplicative) depth $d(f)$.

Because an interactive OT is required for each AND gate, it was believed that GMW is very inefficient compared to Yao's garbled circuits. However, in [CHK+12] it was shown that by using OT extension [IKNP03] and OT pre-computation [Bea95] many OTs can be

pre-computed efficiently in an interactive setup phase. Thereby, all use of symmetric cryptographic operations is shifted to the setup phase, leaving only efficient one-time pad operations for the online phase. Additionally, the setup phase only requires an upper bound on $|f|$ to be known before the secure computation. Follow-up work of [SZ13] demonstrated that, by using OT to pre-compute multiplication triples in the setup phase, the online phase can be further sped up. Multiplication triples are random-looking bits a_i, b_i, and c_i, for $i \in \{\mathcal{A}, \mathcal{B}\}$, satisfying $(c_\mathcal{A} \oplus c_\mathcal{B}) = (a_\mathcal{A} \oplus a_\mathcal{B}) \wedge (b_\mathcal{A} \oplus b_\mathcal{B})$, that are held by the respective parties and used to mask private data during the secure computation. This masking is done very efficiently, since no cryptographic operations are required. In [ALSZ13] it was shown that multiplication triples can be generated interactively using two random OTs. [Hua12] proposed to let a trusted server generate the multiplication triples and send (a_i, b_i, c_i) to party i over a secure channel via the Internet. In our work, we propose to do this locally, without knowing the communication partner in advance.

3 Our Setting

In our setting, depicted in Fig. 2, we focus on efficient ad-hoc secure computation between two semi-honest (cf. §3.1) parties $\mathcal{A}$ and $\mathcal{B}$ who each hold a mobile device, which are approximately equally powerful but significantly weaker than typical desktop computer systems. The parties' devices are connected via a wireless network and are battery-powered.

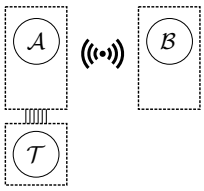


Figure 2: The parties involved in the secure computation.

$\mathcal{A}$ holds a general-purpose tamper-proof hardware token $\mathcal{T}$ that has very few computational resources. $\mathcal{T}$ is powered by $\mathcal{A}$, and its functionalities are limited to the standard functionalities described in §3.2. $\mathcal{A}$ and $\mathcal{T}$ are connected via a physical low-bandwidth connection and communicate via a fixed interface. $\mathcal{B}$ and $\mathcal{T}$ communicate via $\mathcal{A}$, i.e., every message that $\mathcal{B}$ and $\mathcal{T}$ exchange, is seen and relayed by $\mathcal{A}$. Note that this directly requires all communication between $\mathcal{B}$ and $\mathcal{T}$ to be encrypted such that it cannot be read by $\mathcal{A}$. We assume that $\mathcal{T}$ behaves semi-honestly, and is issued by a third party, external to and trusted by both $\mathcal{A}$ and $\mathcal{B}$ (cf. §3.2).

3.1 Adversary Model

We assume that both parties behave semi-honestly in the online phase, i.e., they follow the secure computation protocol, but may try to infer additional information about the other party's inputs from the observed messages. To the best of our knowledge, all previous work on secure computation between two mobile phones is based on the semi-honest model (cf. §7.1). The semi-honest model is suitable in scenarios where the parties want to prevent inadvertent information leakage and for devices where the software is controlled by a trusted party (e.g., business phones managed by an IT department) or where code attestation can be applied. Moreover, this model gives an estimate on the achievable performance of secure computation. We outline how to extend our protocol to malicious security in the full version [DSZ14].

3.2 Trusted Hardware Token

We use the term trusted hardware token $\mathcal{T}$ to refer to a tamper-proof, programmable device, such as a Java smartcard, that offers a restricted set of functionalities. Such functionalities include, for instance, hashing, symmetric and asymmetric encryption/decryption, secure storage and secure random number generation. A detailed summary of standard smartcard functionalities is given in [HL08]. The hardware token is passive, i.e., it cannot initiate a communication by itself and only responds to queries from its host. It contains both persistent and transient memory. $\mathcal{T}$ is physically protected against attacks and is securely erased if it is opened by force. Each token holds an asymmetric key pair, similar to an endorsement key used in TPMs [TCG13], where the public key is certified by a known trusted third party and allows unique identification of $\mathcal{T}$.

Tiny Trusted Third Party $\mathcal{T}$ acts as a tiny trusted third party that behaves semi-honestly. This assumption is similar to the TPM model that is widely used in desktop environments. $\mathcal{T}$ only provides correlated randomness that is later used in the secure computation and does never receive any of $\mathcal{A}$ or $\mathcal{B}$'s private inputs. We assume that only certified code is allowed to be executed on $\mathcal{T}$, and that $\mathcal{T}$ can only actively deviate from the protocol if the hardware token's manufacturer programmed it to be malicious. We assume the code certification was carried out by a trusted third party, and argue that both the manufacturer and the certification authority would face severe reputation loss if it was discovered that they built backdoors into their products. Moreover, we assume that neither $\mathcal{A}$ nor $\mathcal{B}$ colludes with the hardware token manufacturer. This non-collusion assumption is a common requirement for outsourced secure computation schemes

such as [Hua12, KMR12, CMTB13] and enables the construction of efficient protocols. Finally, note that, although $\mathcal{T}$ is in $\mathcal{A}$'s possession, $\mathcal{A}$ cannot easily corrupt $\mathcal{T}$ or obtain its internal information, since $\mathcal{T}$ is assumed to be tamper-proof and does not reveal internal secrets, i.e., the costs of an attack are higher than the benefits from breaking $\mathcal{T}$'s security. This assumption also holds if $\mathcal{A}$ colludes with or impersonates $\mathcal{B}$.

Protection Against Successful Hardware Attacks A malicious adversary could try to break into the hardware token. If such an attack is successful, the following standard countermeasures can be used to prevent further damage: A binding between token and key pair can be realized by using techniques such as physically uncloneable functions (PUFs), however, we are not aware of solutions that are available in commercial products. To bind a token to a certain mobile device or person, $\mathcal{T}$'s certificate could be personalized with one or multiple values that are unique per user and that can be verified over an off-band channel, such as the user's telephone number or the ID of the user's passport. Another line of defense can be certificate revocation lists (CRLs) that allow the users to check if a token is known to be compromised or malicious.

4 Token-aided Mobile GMW

In the following section, we give details on our token-aided GMW-based protocol on mobile devices. Our goal is to minimize the ad-hoc time, i.e., the time from establishing the communication channel between $\mathcal{A}$ and $\mathcal{B}$ until receiving the results of the secure computation. We consider the init phase to not be time critical, but we try to keep its computational overhead small.

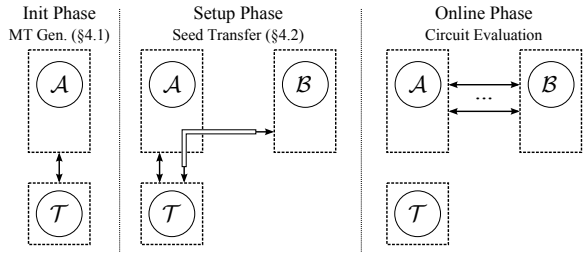


Figure 3: The three phases, workload distribution, and communication in our token-aided scheme.

An overview of our protocol is given in Fig. 3. The general idea is to let the hardware token generate multiplication triples from two (or more) seeds in the init phase that are independent of the later communication partner (§4.1). In the setup phase, $\mathcal{T}$ then sends one seed to $\mathcal{A}$ and the other seed over an encrypted channel to $\mathcal{B}$ (§4.2). The token thereby replaces the OT protocol in the setup phase and allows pre-computing the multiplication triples independently of the communication partner. The online phase of the GMW protocol remains unchanged. In order to overcome the restriction that the function size needs to be known in advance, we describe a method that pre-computes several multiplication triple sequences of different size and only adds a small communication overhead in the setup phase (§4.3). Finally, we analyze the security of our protocol (§4.4) and compare its performance to previous solutions (§4.5).

4.1 Multiplication Triple Pre-Computation in the Init Phase

In the original GMW protocol, $\mathcal{A}$ and $\mathcal{B}$ interactively compute their multiplication triples $(a^n_\mathcal{A}, b^n_\mathcal{A}, c^n_\mathcal{A})$ and $(a^n_\mathcal{B}, b^n_\mathcal{B}, c^n_\mathcal{B})$ in the setup phase using $2n$ random OT extensions (cf. §2.4). Instead, we avoid this overhead in the setup phase and let $\mathcal{T}$ pre-compute the multiplication triples in the init phase as shown in Fig. 4: $\mathcal{T}$ first *generates* random seeds and then *expands* these seeds internally into the multiplication triples and sends $c^n_\mathcal{A}$ to $\mathcal{A}$.

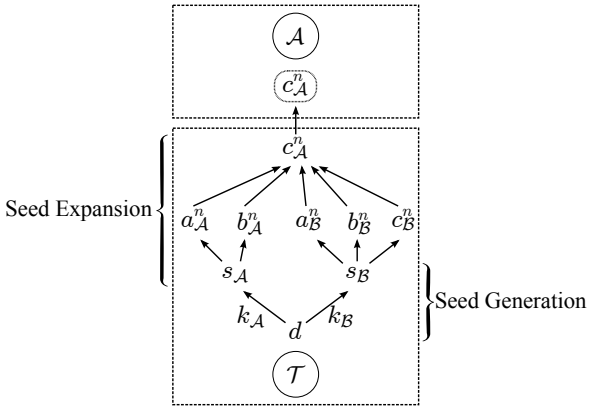


Figure 4: Multiplication triple pre-generation in the init phase between $\mathcal{A}$ and $\mathcal{T}$.

Seed Generation In the seed generation step, $\mathcal{T}$ generates two seeds $s_\mathcal{A} = G_{k_\mathcal{A}}(d)$ and $s_\mathcal{B} = G_{k_\mathcal{B}}(d)$ using a cryptographically strong Pseudo-Random Generator (PRG) G, two master keys $k_\mathcal{A}$ and $k_\mathcal{B}$, and a state value d, which is unique per multiplication triple sequence and can be instantiated with a counter. The two master keys $k_\mathcal{A}$ and $k_\mathcal{B}$ are constant for all multiplication triple sequences and have to be generated and stored only once. Thereby, $\mathcal{T}$ has to store only the unique state value d in its internal memory for every multiplication triple sequence. Note that the only values that will leave the internal memory of $\mathcal{T}$ are the seeds $s_\mathcal{A}$ and $s_\mathcal{B}$

that will be sent in the setup phase to $\mathcal{A}$ and $\mathcal{B}$, respectively (cf. §4.2). In order to ensure that $s_\mathcal{B}$ is not sent out twice, we require $s_\mathcal{A}$ to be queried before $s_\mathcal{B}$ and delete the state value d as soon as $s_\mathcal{B}$ has been sent out. A security analysis of this scheme is given in §4.4.

Seed Expansion The seed expansion step computes a valid multiplication triple sequence from the seeds $s_\mathcal{A}$ and $s_\mathcal{B}$ by computing $(a^n_\mathcal{A}, b^n_\mathcal{A}) = G_{s_\mathcal{A}}(d_\mathcal{A})$ and $(a^n_\mathcal{B}, b^n_\mathcal{B}, c^n_\mathcal{B}) = G_{s_\mathcal{B}}(d_\mathcal{B})$ and setting the remaining value $c^n_\mathcal{A} = (a^n_\mathcal{A} \oplus a^n_\mathcal{B}) \wedge (b^n_\mathcal{A} \oplus b^n_\mathcal{B}) \oplus c^n_\mathcal{B}$, where $d_\mathcal{A}$ and $d_\mathcal{B}$ are publicly known state values of $\mathcal{A}$ and $\mathcal{B}$, respectively. Due to the limited memory of the hardware token, the sequence $c^n_\mathcal{A}$ is computed block-wise such that $\mathcal{T}$ requires only a fixed amount of memory, independently of n, and each block is sent to $\mathcal{A}$, who stores it locally. Note that the values $(a^n_\mathcal{A}, b^n_\mathcal{A}, a^n_\mathcal{B}, b^n_\mathcal{B}, c^n_\mathcal{B})$ do not need to be stored, since they can be expanded from $s_\mathcal{A}$ and $s_\mathcal{B}$, respectively.

4.2 Seed Transfer in the Setup Phase

In the setup phase, the hardware token sends the seeds $s_\mathcal{A}$ and $s_\mathcal{B}$ to $\mathcal{A}$ and $\mathcal{B}$, respectively, and the parties generate their multiplication triples as depicted in Fig. 5. $\mathcal{A}$ obtains his seed $s_\mathcal{A}$ directly from $\mathcal{T}$ and can read the sequence $c^n_\mathcal{A}$, which was obtained in the init phase, from its internal flash storage. $\mathcal{B}$'s seed $s_\mathcal{B}$, on the other hand, cannot be sent in plaintext from $\mathcal{T}$ to $\mathcal{B}$ as the communication between the token and $\mathcal{B}$ is relayed over $\mathcal{A}$, which would allow $\mathcal{A}$ to intercept $s_\mathcal{B}$. We therefore require the communication between $\mathcal{B}$ and $\mathcal{T}$ to be encrypted and $\mathcal{T}$ to authenticate itself to $\mathcal{B}$ with a certificate.

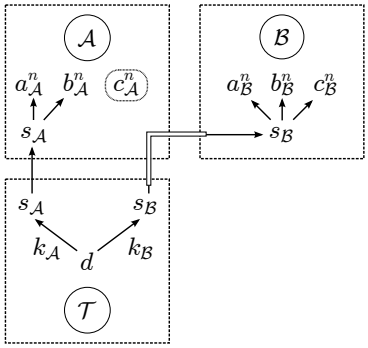


Figure 5: Seed transfer and seed expansion in the setup phase. $s_\mathcal{B}$ is sent from $\mathcal{T}$ to $\mathcal{B}$ over a secure channel.

An encrypted and one-way authenticated communication channel can be established using a key agreement protocol from a wide variety of choices (cf. [MvOV96]). We choose two protocols that allow us to handle different attacker models: For security against a malicious (active) $\mathcal{A}$ we use *TLS* [IET08] (with RSA for public-key crypto, AES for symmetric encryption, and HMAC as message authentication code) and for security against a semi-honest (passive) $\mathcal{A}$ we use KAS1-basic [NIS09] (with AES for symmetric encryption), cf. the full version [DSZ14] for details. Both schemes use $\mathcal{T}$'s public-key certificate that is signed by a trusted third party. For every new connection this certificate is verified by $\mathcal{B}$ and optionally checked against a CRL and/or is checked to be consistent with $\mathcal{A}$'s identity over an out-of-band channel to protect against hardware attacks (cf. §3.2).

4.3 Multiplication Triple Composition

The multiplication triple generation described until now requires the function size $n = |f|$ to be known beforehand. While this may be the case for some functions, e.g., for set intersection using bitwise AND (cf. §6.1), the size of other functions depends on the number of inputs, e.g., the number of contacts in the address book (cf. §6.3). The naive solution to not knowing n in advance would be to generate several multiplication triple sequences of fixed size ℓ in the init phase and send their $\lceil n/\ell \rceil$ seeds in the setup phase, when n is known. However, on average this approach wastes $\ell/2$ multiplication triples and requires to send $\lceil n/\ell \rceil$ multiplication triple seeds. Thus, a smaller ℓ results in fewer wasted multiplication triples but more communication overhead, while a higher ℓ results in more wasted multiplication triples but less communication. Since typical function sizes in secure computation range from millions [HEKM11] to even a billion AND gates [CMTB13], an appropriate ℓ is difficult to choose.

Instead of generating fixed-length blocks of multiplication triple sequences, we propose to generate m multiplication triple sequences $s_0, \ldots, s_{m-1}$ in the init phase, where s_i contains 2^i $(0 \leq i < m)$ multiplication triples. In the setup phase, we then send a set of multiplication triple seeds $\{s_k \,|\, n_k = 1\}$, where n_k is the k-th bit of n. This approach requires sending at most $\lceil \log_2 n \rceil$ seeds. As communication between $\mathcal{T}$ and $\mathcal{A}$ is the bottleneck in our implementation, we set the smallest size of a multiplication triple sequence such that it fits into one packet.

4.4 Security Analysis

In this section, we briefly analyze the security of our protocol for each of the three secure computation phases.

Init Phase In the init phase no private inputs are involved and $\mathcal{B}$ is unknown. Therefore, $\mathcal{A}$ can only try to manipulate the token, which is hard since the hardware token is tamper-proof. Moreover, $\mathcal{A}$ receives only its $c_\mathcal{A}$ shares that do not reveal anything about $\mathcal{B}$'s shares or $\mathcal{T}$'s internal state, due to the cryptographically strong PRG.

Setup Phase The only attack a malicious $\mathcal{A}$ could play in the setup phase, is to impersonate $\mathcal{B}$. This attack is prevented, since every seed s_B can only be queried once (cf. §4.1). The communication between the hardware token and $\mathcal{B}$ is done through an encrypted channel, so that $\mathcal{A}$ cannot get access to those messages. For active security, we use TLS and add a MAC to every packet to prevent modifications and avoid replay attacks. $\mathcal{B}$ cannot actively attack the token since all communication to the hardware token is controlled by $\mathcal{A}$. Obviously, any party can drop or ignore messages, but we exclude this simple denial of service attack from our system model since we assume both parties to be willing to participate in the secure computation. The seeds that each party obtains from the hardware token do also not reveal any additional information since they are directly output from a cryptographically strong PRG to which the hardware token's internal state is used as seed.

Online Phase The security for the online phase directly carries over from the GMW protocol, as we do not introduce any modifications to this phase.

4.5 Performance Comparison

We show that the asymptotic performance of our protocol improves over existing solutions. A summary is shown in Tab. 1 on page 2 and a more detailed comparison is given in the full version [DSZ14]. An experimental evaluation of our protocol is provided in §5.4 and its performance on applications is evaluated in §6.

Asymptotic Performance The init phase of our protocol is, unlike [JKSS10], independent of a concrete instance of f and can thus be pre-computed without knowing a communication partner. During the setup phase, the communication complexity of our protocol is only $O(t)$ (or $\mathcal{O}(t \cdot \log_2 |f|)$ if $|f|$ is unknown), which improves upon the communication of Yao's protocol and the GMW protocol [CHK+12, SZ13, ALSZ13] with $\mathcal{O}(t \cdot |f|)$ communication. Both parties have to do $\mathcal{O}(|f|/b)$ symmetric cryptographic operations to expand their seeds.[2] The online phase is the first phase where f needs to be known. Here, $\mathcal{A}$ and $\mathcal{B}$ send $\mathcal{O}(|f|)$ bits in $d(f)$ rounds, where $d(f)$ is the depth of f. The parties' computation complexity is negligible, as no cryptographic operations are evaluated. This is the biggest advantage over Yao's garbled circuits protocol [MNPS04, JKSS10, HCE11], where $\mathcal{O}(|f|)$ symmetric cryptographic operations have to be evaluated during the online phase.

[2] In our implementation, we instantiate the PRG G with AES-128-CTR, which has block size $b = 128$.

Concrete Performance For 80 bit security, the best known instantiation of Yao's garbled circuits protocol (resp. the GMW protocol) require per AND gate 240 bit (resp. 164 bit) communication and $4+1$ (resp. $12+0$) evaluations of symmetric cryptographic primitives in the setup + online phase. In comparison, our solution requires only 4 bit communication and $0.04+0$ fixed-key AES operations per AND gate.

5 Implementation

This section details the implementation of our scheme. We introduce the smartcard that we use to instantiate the hardware token (§5.1), give an overview of our Android implementation (§5.2), outline our benchmarking environment (§5.3), and experimentally compare the OT extension-based multiplication triple generation to our hardware token-based protocol (§5.4).

5.1 G&D Mobile Security Card

In our implementation we instantiate the trusted hardware token $\mathcal{T}$ with the Giesecke & Devrient (G&D) Mobile Security Card SE 1.0 (MSC). It is embedded into a microSD card that additionally contains 2 GB of separate flash memory. The MSC is based on an NXP SmartMX P5CD080 micro-controller that runs at a maximum frequency of 10 MHz, has 6 kB of RAM, 80 kB of persistent EEPROM, and is based on Java Card version 2.2.2. Note that an applet can only use 1,750 Bytes of the 6 kB RAM for transient storage. The MSC has co-processors for 3DES, AES and RSA that can be called from a Java Card applet, as well as native routines for MD5, SHA-1 and SHA256. The MSC runs the operating system G&D Sm@rtCafe Expert 5.0 which manages the installed Java Card applets, personalization data, and communication keys. The communication between the Android operating system and the MSC is done by a separate service via the SIMalliance Open Mobile API.

5.2 Architecture

The architecture of our implementation is depicted in Fig. 6. To support flexibility and extensibility, our modular architecture consists of the *Application* that specifies the functionality, the *GMWService* that performs secure computation, and the *MTService* that performs the multiplication triple generation and transfer. All communication between $\mathcal{A}$ and $\mathcal{T}$ is done via the MSC Smartcard Service supplied by G&D. The Application can be implemented by a designer and specifies the desired secure computation functionality as a Boolean circuit that can, for instance, be compiled from a high-level circuit description language such as the Secure Function

Definition Language (SFDL) [MNPS04, MLB12] or the Portable Circuit Format (PCF) [KMSB13].

The GMWService implements the GMW protocol and performs the secure computation, given a circuit description and corresponding inputs. The MTService generates the multiplication triples using either OT extension (*OT-Ext*) based on the memory efficient implementation of [HS13] including the optimizations from [ALSZ13] or, if one of the parties holds a hardware token, our token-aided protocol of §4. If a hardware token is present, the MTService manages the multiplication triple generation during the init phase by querying the token and storing the received $c_{\mathcal{A}}$ sequences. For the MSC, the multiplication triple generation on $\mathcal{T}$ is performed via a Java Card applet (*MT JC Applet*) that implements the functionality in §4.1 and is accessible through the Java Card interface. Our implementation can be installed as a regular Android app and does not require root access to the smartphone or a custom firmware.

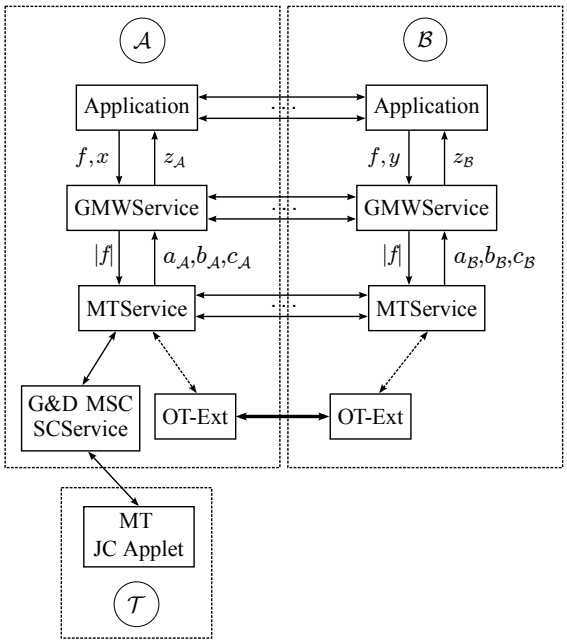


Figure 6: Modular architecture design.

Secure computation is performed by having an Application running on each smartphone, which specifies the function f both parties want to compute securely. From this function the Application generates a circuit description, which it sends to the GMWService. The GMWService interprets the circuit and queries the MTService for the required number of multiplication triples $|f|$. The MTServices on both smartphones then communicate with each other and check whether one of the smartphones holds a hardware token ($\mathcal{A}$ in Fig. 6). If so, both MTServices perform the seed transfer protocol (cf. §4.2), expand the obtained seeds ($\mathcal{A}$ loads the corresponding $c_{\mathcal{A}}$ sequences obtained in the init phase), and merge the obtained multiplication triple sequences (cf. §4.3). If no hardware token is present, the MTServices generate the multiplication triples by invoking OT extension. The MTService then provides the multiplication triples (a_i, b_i, c_i) for $i \in \{\mathcal{A}, \mathcal{B}\}$ to the GMWService. Finally, the Applications send their inputs x and y, respectively, to the GMWService, which performs the secure computation and returns the output $z_i = f(x,y)$.

5.3 Benchmarking Environment

For our mobile benchmarking environment we use two Samsung Galaxy S3's, which each have a 1.4 GHz ARM-Cortex-A9 Quad-Core CPU, 1 GB of RAM, 16 GB of internal flash memory, a mircoSD card slot, and run the Android operating system version 4.1.2 (Jelly Bean). For the communication between the smartphones, we use Wi-Fi direct. For the evaluation, we put the smartphones next to each other on a table. The G&D mobile security card is connected to the mircoSD card slot of one of the phones. We use the short-term security setting recommended by NIST [NIS12], i.e., a symmetric key size of 80 bits and a public key size of 1,024 bit with a 160 bit subgroup. We instantiated the pseudo-random generator G that is used for seed expansion (cf. §4.1) with AES-128 in CTR mode. The hardware token generates multiplication triple sequences of size 2^m for $11 \leq m \leq 24$. We used $m = 11$ as lower bound on the size, since 2,048 is the biggest size we can transfer from $\mathcal{T}$ to $\mathcal{A}$ with a single packet, and $m = 24$ as upper bound, since it was appropriate for our case studies in §6. Finally, we point out that our implementation is single-threaded and utilizes only one of the four available cores of our smartphones. We leave the extension to multiple threads as future work.

5.4 Performance Evaluation

First, we want to quantify the runtime differences between the mobile and the desktop environment. We measure the execution time for AES-128 in ECB mode for an identical single-threaded Java implementation in both domains. The smartphone version is running with 5.5 MB/s while the desktop version achieves 61.1 MB/s. The optimized AES-256 implementation of Truecrypt[3], written in C/C++ and assembly, achieves 143.1 MB/s on the same desktop machine, running without parallelization. For comparison, the smartcard (cf. §5.1) is running AES-128 at a maximum speed of 16.7 KB/s.

In the following we evaluate the performance of our token-based scheme (cf. §4) on smartphones, using TLS or KAS1-basic as key agreement protocol, and compare

[3] http://www.truecrypt.org

it to the OT extension based multiplication triple generation. In our evaluations we only include the time for init and setup phase, since the online phase is identical for both approaches. Results for the online phase are given in §6. All values are averaged over 10 measurements.

Fig. 7 gives an overview over the timings for the generation of 2^m ($11 \leq m \leq 24$) multiplication triples using either OT extension in the setup phase or the hardware token (§4.1) in the init phase. Additionally, the setup phase using TLS and KAS1-basic is depicted, which includes the seed transfer and the seed expansion of $\mathcal{B}$. We always assume the worst case number of seeds to be transferred, i.e., for 2^{24} multiplication triples, we transfer $24-10=14$ seeds (cf. §4.3). Both axes in Fig. 7 are given in a logarithmic scale.

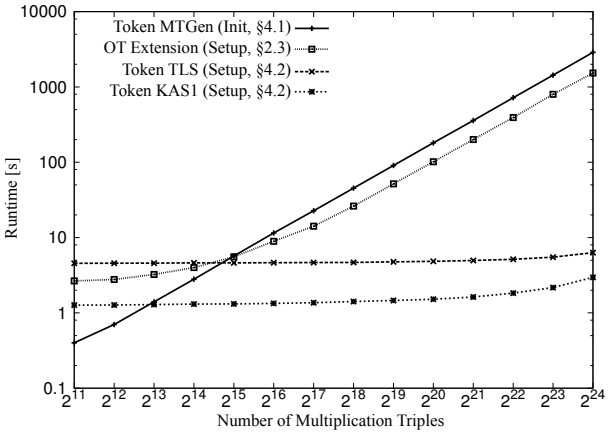


Figure 7: Performance evaluation of the multiplication triple generation and setup phase.

We observe that OT extension on mobile devices is able to generate 2^{24} multiplication triples in 1,529 s, corresponding to 10,971 multiplication triples per second. We ran the same code on two desktop PCs with a 2.5 GHz Intel Core2Quad CPU and 4 GB RAM, connected via Gigabit LAN and were able to compute 2^{24} multiplication triples in 139 s, which indicates a performance decrease of factor 11. While the performance decrease on mobile devices compared to desktop computers was significantly less than the factor of 1000 observed in [HCE11], it is still insufficient for efficiently computing complex functions such as private set-intersection, which typically requires millions of OTs.

In comparison, the multiplication triple generation of the hardware token during the init phase is able to generate 2^{24} multiplication triples in 2,883 s, corresponding to 5,819 multiplication triples per second. For the hardware token-based protocol we observe that the times for sending the seeds using the TLS and KAS1 key agreement protocols grow very slowly with the number of multiplication triples, since the amount of data to be encrypted and sent grows only with $\log_2 |f|$. Additionally, the TLS-based key agreement protocol (4.6 s for 2^{11} multiplication triples) is around factor 3 slower than the KAS1-based key agreement (1.3 s for 2^{11} multiplication triples).

The overall computation and communication workload of OT extension is substantially larger than in our token-based scheme, but its multiplication triple generation rate is not much faster. This can be explained by the faster processing power of the smartphones compared to that of $\mathcal{T}$ and the higher bandwidth of Wi-Fi direct compared to the relatively slow communication channel between $\mathcal{A}$ and $\mathcal{T}$. However, OT extension suffers from high energy consumption, due to the CPU utilization incurred by the symmetric cryptographic operations, as well as the Wi-Fi direct communication [PFW11].

We use PowerTutor[4] to measure the energy consumption of the smartphone's CPU for generating 2^{19} multiplication triples and compare the interactive evaluation of random OT extensions with our smartcard solution. Note that Fig. 8 only displays the CPU's energy consumption whereas the energy consumption of Wi-Fi and the smartcard is not included. However, we argue that the energy consumption of the smartcard is not a critical factor, since these operations can be performed when the phone is charging. The Wi-Fi connection, on the other hand, is required for OT during the setup phase, thus increasing the already high battery drain even further. Moreover, the OT computations have to be done on both devices simultaneously, draining both devices' batteries. Therefore, our token-based solution is particularly well-suited for the mobile domain, where energy consumption and battery lifetime are critical factors.

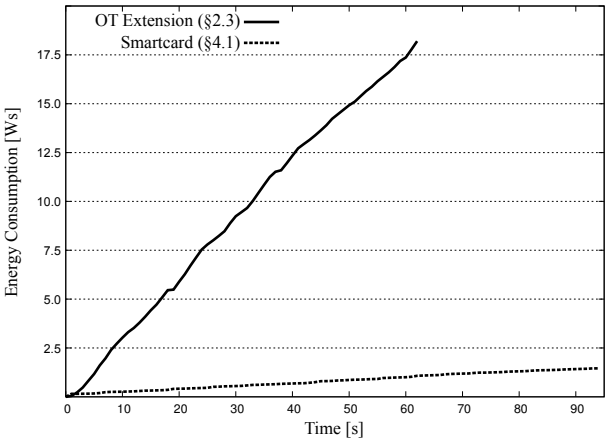


Figure 8: Accumulated smartphone CPU energy consumption during the generation of multiplication triples.

[4] http://powertutor.org

6 Applications

To evaluate the performance of our protocols, we use the mobile phones and setting as specified in §5.3 and consider the following privacy-preserving applications: availability scheduling (§6.1), location-aware scheduling (§6.2), and set intersection (§6.3). We implemented the applications and depict the performance results for an average of 10 iterations. We use KAS1-basic [NIS09] as key authentication scheme. We pre-generated the circuits using the framework of [SZ13], wrote them into a file, and read them on the smartphone. The time for reading the circuit file is included in the setup phase.

6.1 Availability Scheduling

Privacy-preserving availability scheduling is a common example for secure computation on mobile devices [HCC+01, BJH+11] and enables $\mathcal{A}$ and $\mathcal{B}$ to find a possible time slot for a meeting without disclosing their schedules to each other. To schedule a meeting, $\mathcal{A}$ and $\mathcal{B}$ specify a duration and time frame for the meeting. Each party $i \in \{\mathcal{A},\mathcal{B}\}$ then divides the time frame when the meeting can take place (e.g., a week) into n time slots $t_i^n = (t_{i,1},\ldots,t_{i,n})$ and denotes each time slot $t_{i,j} \in \{0,1\}$ as either free ($t_{i,j}=1$) or occupied ($t_{i,j}=0$). The parties compute their common availability t_{Avail}^n by computing the bitwise AND of their time slots, i.e., $t_{Avail}^n = t_{\mathcal{A}}^n \wedge t_{\mathcal{B}}^n$. Overall, this circuit has n AND gates and depth 1. Note that the bitwise AND circuit performs a general functionality and can, for instance, be used for privacy-preserving set intersection where elements are taken from a small domain [HEK12] or location matching [CADT13]. For our experiments, we set the time frames s.t. meetings can be scheduled between 8 am and 10 pm for one day divided into 15 minute slots ($n=56$ slots), one week divided into 15 minute slots ($n=392$ slots), and one month divided into 10 minute slots ($n=2{,}604$ slots). We depict our results in the upper half of Tab. 2.

The multiplication triple generation in the init phase can be performed in several hundred milliseconds, since it requires only one (for 56 and 392 time slots) or two (for 2,604 time slots) packet transfers between $\mathcal{T}$ and $\mathcal{A}$. The setup phase, more detailed the seed transfer protocol, is the main bottleneck in this application, as $\mathcal{T}$ has to perform asymmetric and symmetric cryptographic operations. Finally, the online phase requires only milliseconds but has a high variance, due to the communication over Wi-Fi direct and the small number of communication rounds that are performed in the online phase.

For comparison, we evaluated the same circuit using the mobile Yao implementation of [HCE11] on the same phones, which took factor 1.6 (for the day time frame) up to factor 12 (for the month time frame) longer, cf. Tab. 2.

Table 2: Performance for availability and location-aware scheduling. $|f|$ is the size of the circuit and $d(f)$ its depth. All values measured on smartphones (cf. §5.3).

Time Frame	Day	Week	Month
Availability Scheduling §6.1			
$\|f\|$ / $d(f)$	56 / 1	392 / 1	2,604 / 1
Init [s]	0.37 (±1.6%)	0.37 (±1.6%)	0.73 (±1.0%)
Setup [s]	1.3 (±13%)	1.3 (±13%)	1.3 (±13%)
Online [s]	0.002 (±150%)	0.003 (±167%)	0.007 (±129%)
Ad-Hoc [s]	1.3 (±13%)	1.3 (±13%)	1.3 (±13%)
Mobile Yao [HCE11]			
Ad-Hoc [s]	2.14 (±7.1%)	3.82 (±4.7%)	15.9 (±2.7%)
Location-Aware Scheduling §6.2			
$\|f\|$ / $d(f)$	39,864 / 69	280,605 / 87	1,872,206 / 106
Init [s]	6.9 (±0.3%)	48.5 (±0.2%)	319.6 (±0.5%)
Setup [s]	1.4 (±7.1%)	1.8 (±7.0%)	4.8 (±4.8%)
Online [s]	0.16 (±35%)	0.82 (±7.4%)	5.9 (±18%)
Ad-Hoc [s]	1.5 (±8.4%)	2.6 (±6.5%)	10.7 (±11%)

6.2 Location-Aware Scheduling

In the following we show that our system can be adapted to compute arbitrary and complex functions. We introduce the location-aware scheduling functionality which extends the availability scheduling of §6.1, s.t. the distance between the users is considered as well. The location-aware scheduling functionality takes into account the user's location in a time slot, computes the distance between the users, verifies if a meeting is feasible, and outputs the time slot in which the users have to travel the least distance to meet each other. We argue that this approach is practical, since such position information are often already included in the users' schedules.

In the location-aware scheduling scheme, we assume that the user $i \in \{\mathcal{A},\mathcal{B}\}$ also inputs the location of the previous appointment P_i and the next appointment N_i and the distances that he can reach from his previous appointment p_i and from his next appointment n_i (cf. Fig. 9 for an example). Such p_i and n_i can be computed in plaintext using the distance between P_i and N_i, the free time until the next appointment and the duration of the meeting. The minimal distance among all time slots where the reachable ranges for $\mathcal{A}$ and $\mathcal{B}$ overlap is selected as final result. If successful, the function outputs the identified time slot and for each user whether he should leave from the location of the previous or next appointment. A detailed description of the functionality is given in the full version [DSZ14]. We evaluate the scheme on the same number of time slots used in §6.1 (day, week, month) and depict the performance in the lower half of Tab. 2.

Compared to availability scheduling, the location-aware scheduling circuit is significantly bigger and requires more communication rounds. When performing the scheduling for a month, the circuit consists of 1.8 million instead of 2,604 AND gates for availability schedul-

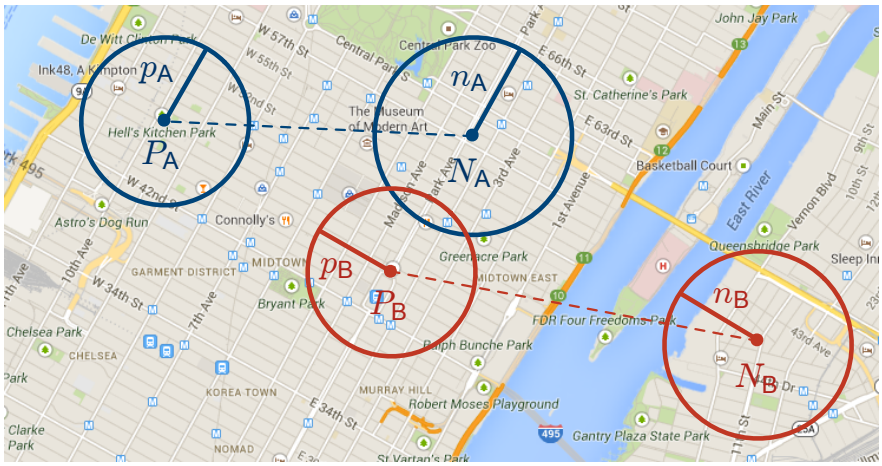


Figure 9: Location-aware scheduling for one time slot of $\mathcal{A}$ and $\mathcal{B}$ with previous locations $P_\mathcal{A}$ and $P_\mathcal{B}$, reachable distances from previous appointments $p_\mathcal{A}$ and $p_\mathcal{B}$, next locations $N_\mathcal{A}$ and $N_\mathcal{B}$ and corresponding reachable distances $n_\mathcal{A}$ and $n_\mathcal{B}$. The meeting can be scheduled between $N_\mathcal{A}$ and $P_\mathcal{B}$ as the reachable ranges overlap.

ing. The time for the init phase increases linearly with the number of AND gates and requires 319 s when performing scheduling for a month. The time for the setup phase is increased less, since the seed transfer grows only logarithmically in $|f|$ and the seed expansion is done efficiently. The online phase is also slowed down substantially (6 s for a month time frame), but is still practical.

6.3 Private Set Intersection

Private set intersection (PSI) is a widely studied problem in secure computation and can be used for example to find common contacts in users' address books [HCE11]. It enables two parties, each holding a set $S_\mathcal{A}$ and $S_\mathcal{B}$ with elements represented as σ-bit strings to determine which elements both have in common, i.e., $S_\mathcal{A} \cap S_\mathcal{B}$, without disclosing any other contents of their sets. While many special-purpose protocols for PSI exist, e.g., [CT10, CT12, CADT13], generic protocols mostly build on the work of [HEK12], where the Sort-Compare-Shuffle (SCS) circuit was outlined. The idea is to have both parties locally pre-sort their elements, privately merge them, check adjacent values for equality, and obliviously shuffle the resulting values to hide their order.

We implement the SCS-WN circuit of [HEK12] which uses a Waksman permutation network to randomly shuffle the resulting elements. We perform the comparison for bit sizes $\sigma \in \{24, 32, 160\}$ and compare the ad-hoc runtime of our protocol to the implementation of [HCE11] for $\sigma \in \{24, 32\}$. The results from [HEK12] are compared to ours for $\sigma \in \{32, 160\}$. The results are given in Fig. 10 and in Tab. 3. Note that [HCE11] and [HEK12] implement Yao's garbled circuits protocol using pipelining, whereas we use the GMW protocol.

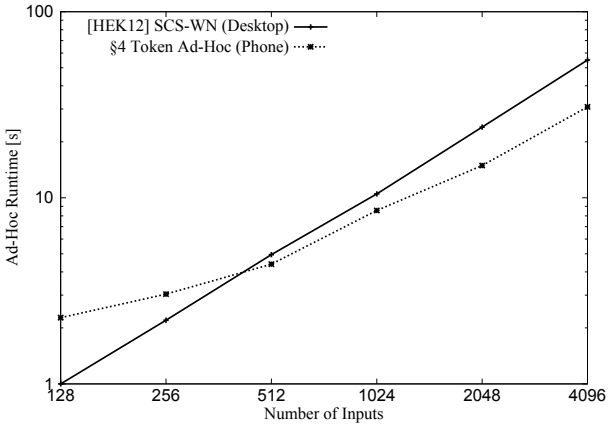


Figure 10: Private set intersection runtime for $\sigma = 32$ bit elements using our token-based protocol on two smartphones (§5.3) and [HEK12] on two desktop PCs.

For a fair comparison, we ran the code from [HCE11] on our Samsung Galaxy S3 smartphones and observed an approximate speedup of factor 2 compared to the measurements from their paper, that were made on older hardware (two Google Nexus One phones). Note that our performance results, as well as the values for the implementation of [HCE11] are benchmarked on mobile devices connected via Wi-Fi Direct, while [HEK12] is benchmarked on two desktop PCs (two Core2Duo E8400 3GHz PCs connected via 100 Mbps LAN).

From Fig. 10 we observe that, due to the seed transfer in our setup phase (cf. §4.2), the Yao's garbled circuits implementation of [HEK12] is faster for up to 256 inputs. However, the seed transfer time amortizes for larger inputs and our token-based scheme outperforms the imple-

Table 3: Ad-hoc runtime of private set intersection where each party inputs n values of σ bits, measured on identical mobile phones (§5.3). [HEK12] results are on PCs and taken from the paper (— indicates that no numbers were given).

Number of Inputs n		32	64	128	256	512	1,024
$\sigma = 24$bit	$\|f\|$	22,432	52,096	118,656	266,240	590,336	1,296,384
	Ours [s]	1.7 (±2.2%)	1.9 (±3.4%)	2.1 (±2.4%)	2.5 (±2.4%)	3.6 (±4.2%)	7.4 (±8.7%)
	[HCE11] [s]	30	68	161	410	1,052	3,010
$\sigma = 32$bit	$\|f\|$	30,368	70,528	160,640	360,448	799,232	1,755,136
	Ours [s]	1.7 (±2.7%)	1.9 (±3.5%)	2.3 (±7.7%)	3.0 (±18%)	4.4 (±9.8%)	8.5 (±20%)
	[HCE11] [s]	42	87	233	565	1,468	4,662
	[HEK12] [s]	—	—	1	2.2	4.95	10.5
$\sigma = 160$bit	$\|f\|$	156,768	364,096	829,312	1,860,864	4,126,208	9,061,376
	Ours [s]	2.2 (±8.8%)	2.7 (±16%)	4.0 (±1.9%)	7.0 (±1.9%)	14.3 (±2.9%)	28.7 (±1.4%)
	[HEK12] [s]	—	—	—	—	—	51.5

mentation of [HEK12], even though our implementation runs on substantially slower mobile phones while theirs is evaluated on two desktop PCs. From Tab. 3 we observe that our scheme outperforms the Yao's garbled circuits implementation of [HCE11], evaluated on identical mobile phones, by factor 18 for 32 inputs with $\sigma = 24$ bit and by up to factor 550 for 1,024 inputs with $\sigma = 32$ bit.

Finally, we compare the performance of our protocol to the PSI protocol of [CADT11,CADT13]. We use their reported numbers for pre-computed PSI on 20 input values and set the bit size $\sigma = 160$ in our protocol.[5] The protocol of [CADT11, CADT13] needs 3.7 s, while our ad-hoc runtime is only 2.1 s (±4.8%). Note, however, that their approach has only a constant number of rounds and can be sped up using multiple cores.

7 Related Work

We classify related work into three categories: secure function evaluation (§7.1), server-aided secure function evaluation (§7.2), and token-based cryptography (§7.3).

7.1 Generic Secure Function Evaluation

The foundations for secure function evaluation (SFE) were laid by Yao [Yao86] and Goldreich et al. [GMW87] who demonstrated that every function that is efficiently representable as Boolean circuit can be computed securely in polynomial time with multiple parties.

SFE Compiler A first compiler for specific secure two-party computation functionalities was presented in [MOR03]. The Fairplay framework [MNPS04] was the first efficient implementation of Yao's garbled circuits protocol [Yao86] for generic secure two-party computation and enabled a user to specify the function to be computed in a high-level language. The FastGC framework [HEKM11] improved on the results of Fairplay by evaluating functions with millions of Boolean gates in mere minutes using optimizations such as the free XOR technique [KS08] and pipelining. The FastGC framework has been used to implement various functions such as privacy-preserving set intersection [HEK12], genomic sequencing, or AES [HEKM11], and was optimized with respect to a low memory footprint in [HS13].

Next to Yao's garbled circuits protocol, the GMW protocol [GMW87] recently received increasing attention. The work of [CHK+12] efficiently implemented GMW in a setting with multiple parties. Subsequently, [SZ13] optimized GMW for the two-party setting and showed that GMW has advantages over Yao's garbled circuits protocol as it allows to pre-compute all symmetric cryptographic operations in a setup phase and that the workload can be split evenly among both parties.

SFE on Mobile Devices A recent line of research aims at making SFE available on mobile devices, such as smartphones. In [HCE11] the authors port the FastGC framework [HEKM11] to smartphones and observe a substantial performance reduction when compared to the desktop environment. They identify the slower processing speed and the high memory requirements as the main bottlenecks. Similarly, [CMSA12] ported the Fairplay framework [MNPS04] to smartphones. A compiler with smaller memory constraints than Fairplay was presented in [MLB12]. We emphasize that previous works on generic SFE on mobile devices use Yao's garbled circuits protocol, whereas our approach is based on GMW.

Several special-purpose protocols for mobile devices using homomorphic encryption were proposed in [BJH+11] (activity scheduling), [CDA11] (scheduling, interest sharing), and [CADT11,CADT13] (comparison, location-based tweets, common friends). In contrast to generic solutions, such custom-tailored protocols can be more efficient, but are restricted to specific functionalities. Their extension to new use cases is complex and usually requires new security proofs.

[5] Note that [CADT11,CADT13] also support bigger bit sizes, since they operate on 1,024-bit ElGamal ciphertexts.

7.2 Server-Aided SFE

One way to speed up generic secure computations on resource constrained devices is to outsource expensive operations to one or more servers. In [HS12] a system for fair server-aided secure two-party computation using two servers was introduced. SALUS [KMR12] is a system for fair SFE among multiple parties using a single server. A system that allows cloud-aided garbled circuits evaluation between one mobile device and a server was introduced in [CMTB13] and its efficiency was demonstrated on large-scale practical applications, such as a secure path finding algorithm. Both [CMTB13] and [KMR12] achieve security against malicious adversaries, but require at least one party to be a machine with more computing power than a mobile phone as it evaluates multiple garbled circuits. [Hua12] proposes that a trusted server generates multiplication triples that are sent to both parties over a secure channel, requiring $\mathcal{O}(|f|)$ bits communication. Instead, we propose to replace the server with a trusted hardware token and show that the communication to one party can be reduced to sub-linear complexity. Moreover, they achieve security against malicious adversaries based on [NNOB12]; we sketch how to extend our work to malicious security in the full version [DSZ14].

We consider this line of research as orthogonal to ours, since it focuses on outsourcing secure computations to a powerful but untrusted cloud server. In contrast, we focus on secure computation between two mobile devices where computations are outsourced to a trusted, but resource constrained smartcard locally held by one party.

7.3 Token-Based Cryptography

Another approach is to outsource computations to trusted hardware tokens, such as smartcards. These tokens are typically resource-constrained, but have the advantage of offering a tamper-proof trusted execution environment.

Setup Assumptions for UC Hardware tokens can be used as setup assumption for Canetti's universal composability (UC) framework, as they allow to construct UC commitments, with which in turn any secure computation functionality can be realized, e.g., [Kat07, DNW09, DKMQ11]. These works are mainly feasibility results and have not been implemented yet.

SFE in Plaintext As discussed in [HL08], the trivial solution to performing SFE using hardware tokens would be to have each party send its inputs over a secure channel to the token, which evaluates f and returns the output. A similar approach with multiple tokens, which additionally provides fault tolerance was given in [FFP+06].

When using the hardware token for plaintext evaluation, the performance of the time-critical online phase is limited by the performance of the token, which is typically very low. Moreover, this requires the token to hold all input values in memory, which quickly exceeds its very limited resources.[6] Alternatively, the token could use external secure memory to store inputs and intermediate values, e.g., [IS05, IS10], but this would require symmetric cryptographic operations in the online phase. Additionally, each new functionality would have to be implemented on the token, whereas our scheme is implemented only once and supports arbitrary functionalities.

Specific Functionalities An efficient protocol for private set-intersection using smartcards was presented in [HL08]. This protocol was extended to multiple untrusted hardware tokens in [FPS+11]. An anonymous credential protocol was presented in [BCGS09].

Outsourcing Oblivious Transfer There are several works that use hardware tokens to compute oblivious transfer (OT): [GT08] implemented non-interactive OT using an extension of a TPM, [Kol10] proposed OT secure in the malicious model using a stateless hardware token, and [DSV10] provided non-interactive OT in the malicious model using two hardware tokens.

We outsource the setup phase of the GMW protocol, which previously was done via OT, to the hardware token. Previous works on outsourcing n OTs require the hardware token to evaluate $\mathcal{O}(n)$ symmetric (or even asymmetric) cryptographic operations in the ad-hoc phase. In comparison, our scheme requires $\mathcal{T}$ to evaluate $\mathcal{O}(n/t)$ symmetric cryptographic operations in the init phase and only $\mathcal{O}(\log_2 n)$ symmetric cryptographic operations in the setup phase (cf. the full version [DSZ14]).

8 Conclusion and Future Work

In this work, we demonstrated that generic ad-hoc secure computation can be performed efficiently on mobile devices when aided by a trusted hardware token. We showed how to extend the GMW protocol by such a token, similar to a TPM, to which most costly cryptographic operations can be outsourced. Our scheme pre-computes most of the workload of GMW in an initialization phase, which is performed independently of the later communication partner and without knowing the function or its size in advance. This is particularly desirable as the pre-computation can happen at any time, e.g., when the device is connected to a

[6] The smartcard we use in our experiments has 1,750 Bytes of RAM, which would be completely filled if each party provided 300 inputs of 24 bits length in private set intersection (cf. §6.3).

power source, which happens regularly with modern smartphones. The remaining interactive ad-hoc phase is very efficient and can be executed in a few seconds, even for complex functionalities. We implemented several privacy-preserving applications that are typical for mobile devices (availability scheduling, location-aware scheduling, and set-intersection) on off-the-shelf smartphones using a general-purpose smartcard and showed that their execution times are truly practical. We found that the performance of our scheme is two orders of magnitude faster than that of other generic secure two-party computation schemes on mobile devices and comparable to the performance of similar schemes in the semi-honest adversary model implemented on desktop PCs.

We see several interesting directions for future research. As our scheme is based on the GMW protocol, it can easily be extended to more than two parties, e.g., for securely scheduling a meeting, cf. [CHK+12]. Moreover, our scheme can be modified to also provide security against malicious parties, cf. [Hua12] (we provide more details in the full version [DSZ14]). Another direction might be equipping both mobile devices with a hardware token to further improve efficiency and/or security.

Acknowledgements

We thank the anonymous reviewers of USENIX Security 2014 for their helpful comments on our paper. We also thank Giesecke & Devrient for providing us with multiple smartcards and the authors of [HCE11] for sharing their code with us. This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE).

References

[ALSZ13] G. Asharov, Y. Lindell, T. Schneider, M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Computer and Communications Security (CCS'13)*, p. 535–548. ACM, 2013.

[BCGS09] P. Bichsel, J. Camenisch, T. Groß, V. Shoup. Anonymous credentials on a standard Java card. In *Computer and Communications Security (CCS'09)*, p. 600–610. ACM, 2009.

[Bea91] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, p. 420–432. Springer, 1991.

[Bea95] D. Beaver. Precomputing oblivious transfer. In *Advances in Cryptology – CRYPTO'95*, volume 963 of *LNCS*, p. 97–109. Springer, 1995.

[BJH+11] I. Bilogrevic, M. Jadliwala, J.-P. Hubaux, I. Aad, V. Niemi. Privacy-preserving activity scheduling on mobile devices. In *ACM Data and Application Security and Privacy (CODASPY'11)*, p. 261–272. ACM, 2011.

[CADT11] H. Carter, C. Amrutkar, I Dacosta, P. Traynor. Efficient oblivious computation techniques for privacy-preserving mobile applications. Technical report, Georgia Institute of Technology, 2011.

[CADT13] H. Carter, C. Amrutkar, I. Dacosta, P. Traynor. For your phone only: Custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN)*, 2013.

[CDA11] E. D. Cristofaro, A. Durussel, I. Aad. Reclaiming privacy for smartphone applications. In *Pervasive Computing and Communications (PerCom'11)*, p. 84–92. IEEE, 2011.

[CHK+12] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, D. Rubenstein. Secure multiparty computation of Boolean circuits with applications to privacy in on-line marketplaces. In *Cryptographers' Track at the RSA Conference (CT-RSA'12)*, volume 7178 of *LNCS*, p. 416–432. Springer, 2012.

[CMSA12] G. Costantino, F. Martinelli, P. Santi, D. Amoruso. An implementation of secure two-party computation for smartphones with application to privacy-preserving interest-cast. In *Privacy, Security and Trust (PST'12)*, p. 9–16. IEEE, 2012.

[CMTB13] H. Carter, B. Mood, P. Traynor, K. Butler. Secure outsourced garbled circuit evaluation for mobile phones. In *USENIX Security'13*, p. 289–304. USENIX, 2013.

[CT10] E. De Cristofaro, G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, p. 143–159. Springer, 2010.

[CT12] E. De Cristofaro, G. Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing (TRUST'12)*, volume 7344 of *LNCS*, p. 55–73. Springer, 2012.

[DKMQ11] N. Döttling, D. Kraschewski, J. Müller-Quade. Unconditional and composable security using a single stateful tamper-proof hardware token. In *Theory of Cryptography Conference (TCC'11)*, volume 6597 of *LNCS*, p. 164–181. Springer, 2011.

[DNW09] I. Damgård, J. B. Nielsen, D. Wichs. Universally composable multiparty computation with partially isolated parties. In *Theory of Cryptography Conference (TCC'09)*, volume 5444 of *LNCS*, p. 315–331. Springer, 2009.

[DSV10] M. Dubovitskaya, A. Scafuro, I. Visconti. On efficient non-interactive oblivious transfer with tamper-proof hardware. Cryptology ePrint Archive, Report 2010/509, 2010. http://eprint.iacr.org/2010/509.

[DSZ14] D. Demmler, T. Schneider, M. Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. Cryptology ePrint Archive, Report 2014/467, 2014. http://eprint.iacr.org/2014/467.

[FFP+06] M. Fort, F. C. Freiling, L. D. Penso, Z. Benenson, D. Kesdogan. TrustedPals: Secure multiparty computation implemented with smart cards. In *European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *LNCS*, p. 34–48. Springer, 2006.

[FPS+11] M. Fischlin, B. Pinkas, A.-R. Sadeghi, T. Schneider, I. Visconti. Secure set intersection with untrusted hardware tokens. In *Cryptographers' Track at the RSA Conference (CT-RSA'11)*, volume 6558 of *LNCS*, p. 1–16. Springer, 2011.

[GMW87] O. Goldreich, S. Micali, A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Symposium on Theory of Computing (STOC'87)*, p. 218–229. ACM, 1987.

[GT08] V. Gunupudi, S. R. Tate. Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In *Financial Cryptography and Data Security (FC'08)*, volume 5143 of *LNCS*, p. 98–112. Springer, 2008.

[HCC+01] T. Herlea, J. Claessens, D. De Cock, B. Preneel, J. Vandewalle. Secure meeting scheduling with agenTa. In *Communications and Multimedia Security (CMS'01)*, volume 192 of *IFIP Conference Proceedings*, p. 327–338. Kluwer, 2001.

[HCE11] Y. Huang, P. Chapman, D. Evans. Privacy-preserving applications on smartphones. In *Hot topics in security (HotSec'11)*. USENIX, 2011.

[HEK12] Y. Huang, D. Evans, J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed Security Symposium (NDSS'12)*. The Internet Society, 2012.

[HEKM11] Y. Huang, D. Evans, J. Katz, L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security'11*, p. 539–554. USENIX, 2011.

[HL08] C. Hazay, Y. Lindell. Constructions of truly practical secure protocols using standard smartcards. In *Computer and Communications Security (CCS'08)*, p. 491–500. ACM, 2008.

[HS12] A. Herzberg, H. Shulman. Oblivious and fair server-aided two-party computation. In *Availability, Reliability and Security (ARES'12)*, p. 75–84. IEEE, 2012.

[HS13] W. Henecka, T. Schneider. Faster secure two-party computation with less memory. In *Symposium on Information, Computer and Communications Security (ASIACCS'13)*, p. 437–446. ACM, 2013.

[Hua12] Y. Huang. *Practical Secure Two-Party Computation*. PhD dissertation, University of Virginia, 2012.

[IET08] IETF. The Transport Layer Security (TLS) Protocol Version 1.2. Technical report, Internet Engineering Task Force (IETF), 2008.

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, p. 145–161. Springer, 2003.

[IS05] A. Iliev, S. Smith. More efficient secure function evaluation using tiny trusted third parties. Technical report, Dartmouth College, Computer Science, 2005.

[IS10] A. Iliev, S. W. Smith. Small, stupid, and scalable: secure computing with Faerieplay. In *Workshop on Scalable Trusted Computing (STC'10)*, p. 41–52. ACM, 2010.

[JKSS10] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, T. Schneider. Embedded SFE: Offloading server and network using hardware tokens. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, p. 207–221. Springer, 2010.

[Kat07] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *Advances in Cryptology – EUROCRYPT'07*, volume 4515 of *LNCS*, p. 115–128. Springer, 2007.

[KMR12] S. Kamara, P. Mohassel, B. Riva. Salus: a system for server-aided secure function evaluation. In *Computer and Communications Security (CCS'12)*, p. 797–808. ACM, 2012.

[KMSB13] B. Kreuter, B. Mood, A. Shelat, K. Butler. PCF: a portable circuit format for scalable two-party secure computation. In *USENIX Security'13*, p. 321–336. USENIX, 2013.

[Kol10] V. Kolesnikov. Truly efficient string oblivious transfer using resettable tamper-proof tokens. In *Theory of Cryptography Conference (TCC'10)*, volume 5978 of *LNCS*, p. 327–342. Springer, 2010.

[KS08] V. Kolesnikov, T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, p. 486–498. Springer, 2008.

[MLB12] B. Mood, L. Letaw, K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Financial Cryptography and Data Security (FC'12)*, volume 7397 of *LNCS*, p. 254–268. Springer, 2012.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security'04*, p. 287–302. USENIX, 2004.

[MOR03] P. MacKenzie, A. Oprea, M. K. Reiter. Automatic generation of two-party computations. In *Computer and Communications Security (CCS'03)*, p. 210–219. ACM, 2003.

[MvOV96] A. Menezes, P. C. van Oorschot, S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NIS09] NIST. NIST Special Publication 800-56b, Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization). Technical report, National Institute of Standards and Technology (NIST), 2009.

[NIS12] NIST. NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Rev. 3). Technical report, National Institute of Standards and Technology (NIST), 2012.

[NNOB12] J. B. Nielsen, P. S. Nordholt, C. Orlandi, S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO'12*, volume 7417 of *LNCS*, p. 681–700. Springer, 2012.

[NP01] M. Naor, B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium On Discrete Algorithms (SODA'01)*, p. 448–457. Society for Industrial and Applied Mathematics, 2001.

[PFW11] G. P. Perrucci, F. H. P. Fitzek, J. Widmer. Survey on energy consumption entities on the smartphone platform. In *Vehicular Technology Conference (VTC'11)*, p. 1–6. IEEE, 2011.

[Rab81] M. O. Rabin. *How to exchange secrets with oblivious transfer*, TR-81 edition, 1981. Aiken Computation Lab, Harvard University.

[SZ13] T. Schneider, M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security (FC'13)*, volume 7859 of *LNCS*, p. 275–292. Springer, 2013.

[TCG13] TCG. TCG TPM specifications 2.0, 2013. Trusted Computing Group.

[Yao86] A. C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science (FOCS'86)*, p. 162–167. IEEE, 1986.

ZØ: An Optimizing Distributing Zero-Knowledge Compiler

Matthew Fredrikson
University of Wisconsin

Benjamin Livshits
Microsoft Research

Abstract

Traditionally, confidentiality and integrity have been two desirable design goals that are have been difficult to combine. *Zero-Knowledge Proofs of Knowledge* (ZKPK) offer a rigorous set of cryptographic mechanisms to balance these concerns. However, published uses of ZKPK have been difficult for regular developers to integrate into their code and, on top of that, have not been demonstrated to scale as required by most realistic applications.

This paper presents ZØ (pronounced "zee-not"), a compiler that consumes applications written in C# into code that automatically produces scalable zero-knowledge proofs of knowledge, while automatically splitting applications into distributed multi-tier code. ZØ builds detailed cost models and uses two existing zero-knowledge back-ends with varying performance characteristics to select the most efficient translation. Our case studies have been directly inspired by existing sophisticated widely-deployed commercial products that require both privacy and integrity. The performance delivered by ZØ is as much as 40× faster across six complex applications. We find that when applications are scaled to real-world settings, existing zero-knowledge compilers often produce code that fails to run or even *compile* in a reasonable amount of time. In these cases, ZØ is the only solution we know about that is able to provide an application that works at scale.

1 Introduction

As popular applications rely on personal, privacy-sensitive information about users, factors such as legal regulations, industry self-regulation, and a growing body of privacy-conscious users all pressure developers to respond to demands for privacy. Storing user's data in the cloud creates downsides for the application provider, both immediately and down the road. While policy measures such as DoNotTrack and anonymous advertising identifiers become increasingly popular, a recent trend explored in several research projects has been to move functionality to the *client* [13, 17, 37, 40]. Because execution happens on the client, such as a mobile device or even in the browser, this alone provides a degree of privacy in the computation: only relevant data, if any, is disclosed (to a server). However, in many cases, moving functionality to the client conflicts with a need for computational *integrity*: a malicious client can simply forge the results of a computation.

Traditionally, confidentiality and integrity have been two desirable design goals that are have been difficult to combine. *Zero-Knowledge Proofs of Knowledge* (ZKPK) offer a rigorous set of cryptographic mechanisms to balance these concerns, and recent theoretical developments suggest that they might translate well into practice. In the last several years, zero-knowledge approaches have received a fair bit of attention [23]. The premise of zero-knowledge computation is its promise of both privacy *and* integrity through the mechanism cryptographic proofs. However, published uses of ZKPK [4, 5, 7, 8, 19, 36] have been difficult for regular developers to integrate into their code and, on top of that, have not been demonstrated to scale, as required by most realistic applications.

Zero-knowledge example: pay as you drive insurance: A frequently mentioned application and a good example of where zero-knowledge techniques excel is the practice of *mileage metering* to bill for car insurance: pay as you drive auto insurance is an emerging scheme that involves paying a rate proportional to the number of miles driven, either linearly, or using several billing brackets [4, 38, 41]. Of course, given that the insurance company knows much about the customer, including their address, if daily mileage data is provided, much can be inferred about user's daily activities, where they shop, etc. [15, 29, 30]. The user in this scheme performs a calculation on their own data, but of course the insurance company wants to prevent cheating. Zero-knowledge proofs provide a way to ensure both privacy and integrity, which involves performing the billing computation on the user's hardware (on the *client*), perhaps, monthly, and providing the insurance company with 1) the final bill and 2) a proof of correctness of the accounting calculation, which can be verified by the insurance company (on the *server*) [4, 18, 35, 39].

What we did: In this paper, we present ZØ, a compiler that consumes applications written in a subset of C# into code that produces scalable zero-knowledge proofs of knowledge, while automatically splitting applications into distributed code, to be executed on two (or more)

execution tiers. We are building on very recent developments in zero-knowledge cryptographic techniques [16, 31], exposing to the developer the ability to take advantage of these advances. ZØ builds detailed cost models of the code regions that require ZKPK, and uses existing zero-knowledge back-ends with varying performance characteristics to select the most efficient translation, by formulating and solving constrained numeric optimization problems. Our cost modeling takes advantage of the strengths of both back-ends, while avoiding their weaknesses, both for local and global (distributed) optimization. Using a set of realistic applications that perform tasks such as distributed data mining and crowd-sourced data aggregation, we demonstrate ZØ's ability to produce privacy-preserving code which runs significantly faster than previously possible.

High-level goals: ZØ aims to provide an attractive combination of high-level goals of *privacy*, *integrity*, *expressiveness*, and *performance*. While the first two goals are achieved through the use of zero-knowledge, to support ease of programming and expressiveness, ZØ accepts (a subset of) C#, a widely-used general purpose language as input that can run in many settings. Of course, we are not tied to C# and could support another high-level language such as JavaScript, Java, or C++. Our use of a general-purpose language allows developers to include hundreds or thousands of lines of C# or other .NET code, allowing the construction of full-featured GUI-based distributed applications that support zero-knowledge instead of small examples written in a domain-specific language.

To enable distributed programming wherever .NET code can run, ZØ supports automatic tier-splitting, inspired by distributing compilers such as GWT [20] and Volta [24]. We primarily target client-server computations (two tiers), although other options such as P2P are also supported by ZØ. Code produced by ZØ can be run on desktops, in the cloud, on mobile devices (Windows Phone) and on the web (Silverlight).

Applications: Much of the inspiration for ZØ came from our desire to be able to use ZKPK techniques to build applications directly analogous to some widely-deployed commercial products, as opposed to toy benchmarks. In our studies detailed in Section 7, we show how they can be (re-)built in a privacy- and integrity-preserving way. For example, our FitBit study was inspired by wireless activity tracking devices manufactured by FitBit (`fitbit.com`) and Earndit (`earndit.com`). The Slice study was inspired by purchase tracking software from Slice, Inc. (`slice.com`). The study Waze app was inspired by Waze, a popular crowd-sourced, real-time traffic app for mobile platforms (`waze.com`).

Contributions: We make these contributions:

- This paper proposes ZØ, a distributing compiler that allows developers to create highly performant, large distributed applications, while preserving both privacy and integrity. ZØ uses precisely calibrated *cost models* to choose which underlying zero-knowledge back-end to employ. Based on the cost model, ZØ statically determines the appropriate *splitting perimeter* for the application to achieve best performance and rewrites it to be run on multiple tiers.
- **Developer**: ZØ is designed to be easily accessible to a regular developer; to this end, we expose zero-knowledge functionality via LINQ, language-integrated-queries built into .NET. We demonstrate the expressiveness of the ZØ approach by developing six case studies directly inspired by commercial applications which we hope will become benchmarks for zero-knowledge tools, ranging from personal fitness tracking (Fitbit) to crowd-sourced traffic-based routing (Waze), to personalized shopping scenarios.
- **Cost modeling**: We develop cost models for the individual back-ends, allowing us to perform global cross-tier optimizations. Our cost-fitting models provide an excellent match with the observed performance, with R^2 scores between .98 and .99.
- **Speedup**: We evaluate ZØ on six complex real-life large-scale applications of zero knowledge, focusing on latency and throughput of zero-knowledge tasks. Our global optimizer is fast, completing in under 3 seconds on all programs. ZØ produces code that achieves as much as 40× speedups compared to state-of-the art zero-knowledge systems. We also find that ZØ is able to effectively optimize *across tiers* in a distributed application: while the code it generates may be slower on one tier (we observed one case that was 2× slower for the server), the savings at other tiers are always greater (e.g., 4× faster on the client).
- **Scale**: At scale, existing zero-knowledge compilers often produce code that fails to run in a reasonable amount of time, or exhaust system resources during compilation. In these cases, ZØ is the *only* solution that is able to provide a working application.

Paper Organization: The rest of the paper is organized as follows. Section 2 provides motivating examples and some background on zero-knowledge. Section 3 gives an overview of the ZØ approach. Section 5 describes the ZØ compiler implementation. Section 4 talks about cost models and both local and global optimizations ZØ performs. Section 5 describes ZØ implementation. Section 6 discusses how ZØ translates C# into ZK proof-generating code. Section 7 presents six case studies. Section 8 describes our experimental evaluation. Related work is discussed in Section 10 and Section 11 concludes the paper. The PDF version of this paper has

```
public class LoyaltyCard : DistributedRuntime
{
  // Local variable declarations
  [Location(Client)] IEnumerable<int> shophist;
  [Location(Client)] IEnumerable<int> items;
  IEnumerable<Triple> automaton;
  IEnumerable<Pair> transducer;

  public void Initialize(string[] args)
  {...}

  public void DoWork(string[] args)
  {
    var discount =
      GetDiscounts(shophist, items,
                   automata, transducer);
    ApplyDiscount(discount);
  }

  [Location(Client)]
  IEnumerable<Pair> GetDiscounts(
   [MaxSize(Purchases)] IEnumerable<int> history,
   [MaxSize(Items)] IEnumerable<int> items,
   [MaxSize(Edges)] IEnumerable<Triple> automata,
   [MaxSize(States)] IEnumerable<Pair> transducer)
  {
    ZeroKnowledgeBegin();
    // Check that the history is in ascending order
    var historyAscendingCheck = history.Aggregate(
      0,
      (last, curel) => check(last <= curel));
    // Get the "discount state"
    var purch_state = history.Aggregate(
      0,
      (state, purch) =>
        automaton.First(
         trans => (trans.fld(1) == state) &&
                  (trans.fld(2) == purch)).
                                    fld(3));
    var discount = history.Aggregate(
      new Pair(purch_state, 0),
      (state, purch) =>
        new Pair(
         // Get the next automata state
         automata.First(
          trans => (trans.fld(1) == state.fld(1))
                   && (trans.fld(2) == purch)).
                   fld(3),
         // Total the current state discount
         state.fld(2) + transducer.First(
           edge => edge.fld(1) == state.fld(1)));
    ZeroKnowledgeEnd();

    return new IEnumerable<Pair>(discount);
  }

  [Location(External)] void ApplyDiscount(...)
  {...}
}
```

Figure 1: Example application: a personalized retail loyalty card.

an with additional diagrams to supplement the main text.

2 Background

To explain the goals of ZØ concretely, we will demonstrate its functionality on a smartphone application with conflicting privacy and integrity needs.

2.1 Example: Retail Loyalty Card

Figure 1 shows the ZØ code for a personalized retail loyalty card mobile app, with functionality similar to Safeway's "Just for U" application or Walgreens' iOS application. Each time the customer reaches the check-out line, this application interacts with the retail terminal in a bi-directional exchange of information. The exchange takes place using the phone's built-in NFC sensor.

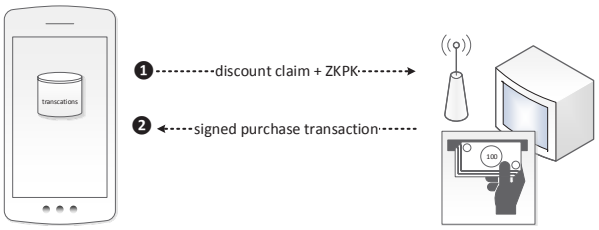


Figure 2: Personalized loyalty card application.

First, the application sends a *discount claim* to the retail terminal, pertaining to the items the customer is about to purchase. This discount is computed based on the customer's previous purchases, using personalization to provide enhanced value and incentive for the customer. Zero-knowledge proofs are supplied to ensure the privacy of the customer's shopping history, without sacrificing the trustworthiness of their discount claim.

Second, the terminal sends a list of purchases to the client, corresponding to the current check-out transaction. This list, along with the customer's other previous purchases, will be stored in a client-side database used to compute a discount the next time the user shops with this retailer.

Application Code: Figure 1 contains C# code for computing the core functionality of this application: using the customer's purchase history to produce a discount, and sending that discount to the retail terminal. It is important to notice that this is standard C#, capable of seamless incorporation into larger bodies of C# code. In fact, ZØ extends the standard C# compiler, and only applies specialized reasoning to classes that inherit from ZØ's `DistributedRuntime` class. All of the UI and external library code can remain in the application, without affecting the performance and functionality of ZØ. This allows ZØ to scale to large applications with arbitrary legacy dependencies, provided that the sections requiring zero-knowledge reasoning are localized and moderate in size. Several important points bear mentioning.

First, of the four functions, two of them, which we call *worker functions*, contain *location annotations*: `GetDiscounts` is constrained to execute on the client (e.g., the user's smartphone), and `ApplyDiscount` to `External` (e.g., the retail terminal). ZØ generates separate object code for each of these locations, and inserts code to handle the network transfer and data marshalling for any dependencies between these two functions. In order to streamline the code generated by ZØ, the worker functions must always return `void` or `IEnumerable` ob-

jects, which ZØ's underlying runtime is optimized to quickly marshall and transfer.

Second, the target functionality is computed from the main function `DoWork`, which is called after `Initialize`. `Initialize` gives the application an opportunity to prepare the class's local state by reading sensors, buffering data, etc., and can contain arbitrary C# code. `DoWork` is more constrained: it can contain a sequence of calls to worker functions, with no intermediate local computations, branching statements, or loop statements. This allows ZØ to efficiently compute the dependencies between different tiers. In this case, ZØ determines that the return value of `GetDiscounts` (computed on the smartphone) is always used by `ApplyDiscount` (computed on the retail terminal), and inserts code to package and send, or receive and unpack, the necessary data as well as any accompanying zero-knowledge proofs.

Third, the main code is located in `GetDiscounts`, which takes a list of the user's previous purchases (`history`), the user's current check-out items (`items`), and a finite-state transducer (`automata` and `transducer`), and produces a discount dollar value for transfer to the retail terminal. The transducer is produced by the retailer, and is designed to associate past purchases to items that the customer may be interested in buying in the future; the details of designing the transducer are beyond the scope of this work. `GetDiscounts` begins by checking that the purchases are given in ascending order, by their ID numbers; this is a simple optimization that allows the retailer to minimize the size of the transducer. This check is performed using LINQ's `Aggregate` operator, and ZØ's `check` function, which behaves like an assertion. It then proceeds to traverse the transducer's finite-state machine using the customer's shopping history, effectively loading the history into the transducer's memory in preparation for emitting discount values.

Finally, the customer's current items are processed by traversing the finite-state machine, starting in the final state of the previous traversal, and summing the output of the transducer relation. The final sum is returned to `DoWork` as a discount claim.

Zero-knowledge: The entirety of `GetDiscounts` is computed in zero-knowledge, as indicated by the `ZeroKnoweldgeBegin`() and `ZeroKnowledgeEnd`() annotations. Notice that each statement of this method consists of a LINQ query, giving the computation an overall functional form, without using language features such as references, loops, or conditionals. This is necessary to accommodate faithful translation into code that produces zero-knowledge proofs using the zero-knowledge backends discussed in Section 2.2. However, the programmer is still able to express computations in this fragment of standard C#, without dealing with the overhead of inter-language binding between the engines and the main program, and without needing to learn the different input languages understood by each engine.

Finally, a few subtle details of this code bear mentioning. Two of the class variable declarations, `shophist` and `items`, have location annotations that tell ZØ that they should not leave the customer's smartphone without first being processed by zero-knowledge code. This gives the programmer an extra degree of assurance of the code's privacy properties, letting her treat the zero-knowledge code regions like *declassifiers* with additional integrity guarantees. Finally, notice that the parameters to `GetDiscounts` contain `MaxSize` attribute annotations. These optional size annotations allow the ZØ compiler to do precise cost modeling, as explained in Section 4.

2.2 Zero-Knowledge Back-ends

ZØ relies on two zero-knowledge back-ends, Pinocchio [31] and ZQL [16], to produce code that balances privacy and integrity. Each of these back-ends takes an expression, in the form of executable code in a high-level source language, and produces object code that computes the expression over dynamically-provided inputs while building zero-knowledge proofs for the expression on the given input. These engines have very different characteristics that affect performance and usability in different ways, which we outline here.

Pinocchio: Pinocchio utilizes a novel underlying computation model, *Quadratic Arithmetic Polynomials*, to evaluate an expression and produce zero-knowledge proofs [31]. For some computations, it yields performance gains several orders of magnitude beyond previous systems that gave similar functionality, producing proofs of a *constant size* regardless of the size or structure of the target expression.

The expression language supported by Pinocchio is a strict subset of C, and the object created for evaluation is an *arithmetic circuit* [31]. The fact that the target circuit must be finite, and cannot encode *side-effects*, imposes necessary conditions on the parts of C that are available. Loops and conditionals are "unrolled" during compilation, so all loops must have static bounds. Likewise, pointers and array indices must be compile-time constants, or simple loop variables (as these are unrolled), thus simplifying cost modeling. For this paper we used a publicly released version of Pinoccio 0.4 obtained from the public distribution[1].

ZQL: ZQL utilizes several fairly recent advances in the theory of zero-knowledge proofs to produce efficient verified private code that operates over functional lists [16]. The underlying cryptographic machinery used by ZQL is more traditional than that of Pinocchio, relying heavily on homomorphic commitment schemes to provide its

[1] `https://vc.codeplex.com/downloads/get/714129`

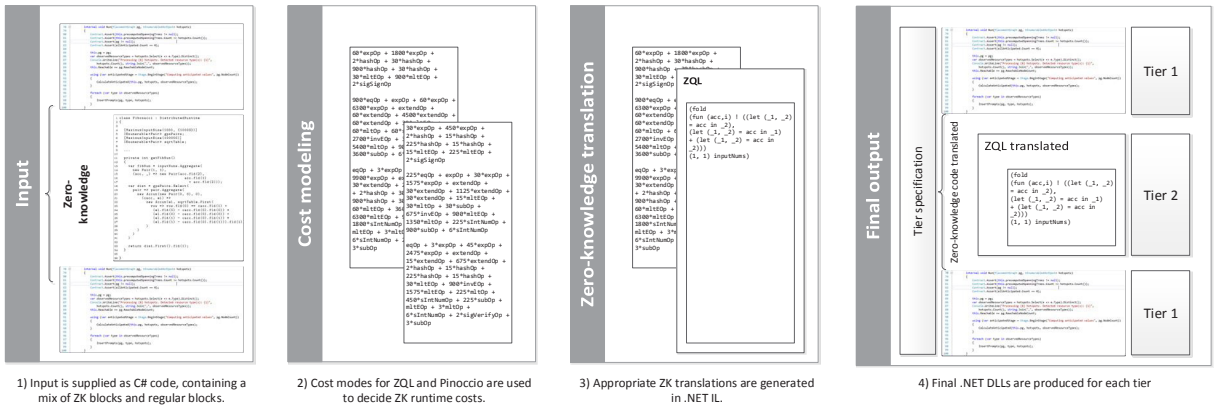


Figure 3: ZØ architecture. ZQL and Pinoccio are used as sample back-ends for illustrative purposes.

guarantees. The expression language supported by ZQL is a simple functional language without side effects, and limited operator support. In a nutshell, ZQL supports map and fold operations, as well as find operations over tuples of integers. Boolean expressions can only be used inside of find operations, and are currently limited to conjunctions of equality tests; all forms of inequality are not explicitly supported, although the authors plan to support these operations in future versions. In terms of arithmetic, addition, subtraction, and multiplication are supported. Finally, multiple operations can be sequenced using classic functional let bindings. Although these constructs might seem modest at first blush, the ability to perform table lookups using find allows for the evaluation of logic gates, and the list-based map and fold operations place no upper-bound on the size of the program's input, as in the case of Pinocchio. We obtained a version of ZQL from its authors.

3 Overview

Figure 3 shows the architecture of the ZØ compiler. The developer provides as input a set of C# source files, which may include arbitrary regions of legacy and library code as well as functionality targeted towards zero-knowledge proof generation. ZØ then enters a *cost modeling* stage, analyzing the zero-knowledge regions, building performance models that characterize the cost of providing zero-knowledge proof generation and verification code for each available zero-knowledge back-end. These models take the form of polynomials over the size of the input data to the zero-knowledge region in the original C# application. ZØ then compares the models to determine which engine the application should use for each C# statement in the region, and translates the C# code (depicted in the *zero-knowledge translation* stage of Figure 3) into expressions understood by the appropriate zero-knowledge engine. In the *final output* stage (Figure 3), ZØ decides how to split the application across tiers to maximize performance, given privacy annotations as well as relative *costs* for transmitting data and computing at each tier.

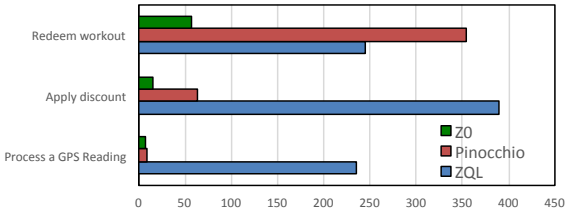


Figure 4: Comparison of times for several applications.

This translation yields a separate module which is callable from the original application, either as an *arithmetic circuit* (Pinocchio) or standard .NET bytecode (ZQL). Finally, ZØ partitions the original C# code, along with the zero-knowledge modules compiled in the previous step, into multiple applications to run at each *service tier*. During partitioning, ZØ inserts code to perform communication, synchronization, data marshaling, and zero-knowledge proof transfer in parallel to the original application code. The resulting modules are standard .NET bytecode that can be run on the proper tiers without the need for additional specialized software.

Optimization & cost models: Even apparently straightforward applications like the personalized loyalty card app discussed in Section 2.1 contain subtle characteristics that might make zero-knowledge proof generation expensive. It is often the case that one zero-knowledge engine offers significantly better performance for a particular statement, and selecting the appropriate engine for each computation in the zero-knowledge region means the difference between a scalable, low-latency implementation and one that requires hours or days to execute.

For the loyalty card application in Figure 1, it turns out that the inequality comparisons are better handled by Pinocchio, whereas the table lookups needed to execute the transducer are very inexpensive when performed by ZQL. A comparison of the times to perform the operation on the y-axis for several applications from Section 7 is shown in Figure 4. We can see dramatic differences in performance between the back-ends, with the ZØ approach out-performing either of the two back-ends. ZØ addresses these performance differences by building detailed performance models for each statement in the zero-knowledge region.

Distributed configuration: To support a variety of distributed scenarios, ZØ allows the developer to place code on several different tiers, which are specified using the following *tier labels*: Client (end-user's primary device), External (provider's servers), ClientShare (peer-to-peer nodes), and ClientResource (additional hosts owned by end-user). Tiers impose data confidentiality and integrity constraints, as ZØ makes assumptions about the *trust relationships* between tiers.

The figure in this paragraph shows these relationships; white cells indicate trust, and gray the opposite. At compile time, the user can modify the configuration by specifying *weights* on each tier label indicating the relative cost of computation at that tier, as well as the cost of communication between tiers. ZØ uses these weights during optimization to determine the best placement of code and data amongst the tiers, and are only necessary to fine-tune the performance of certain applications; they can be ignored and left at the default value of 1 by default. Data privacy constraints are given by the programmer by marking certain variables as *private* to a particular tier using the attribute [Private(T_L)], where T_L specifies the tier to which the data is considered private (e.g., Client, External, ...).

	C	CS	CR	E
C				
CS				
CR				
E				

Note that by design, these annotations are lightweight: they are only needed on (the few) variables that must be kept confidential. Most can be declared without any annotations at all.

When ZØ compiles the application and runs a global optimization described in Section 4.2 to place each worker method on a specific tier, privacy annotations are used in part to determine on which tiers a method may reside. These constraints are *hard*, meaning that a privacy annotation that requires a less performant compilation configuration will always be respected; if the privacy constraints conflict with each other, then compilation will not terminate early. Privacy annotations are propagated transitively using a local dataflow analysis, so that dependent variables have matching annotations.

Threat model: Because of its reliance on zero-knowledge back-ends, ZØ makes all of the assumptions needed for security by ZQL [16] and Pinocchio [31]. The result of ZØ compilation will be executed on one or more tiers. Privacy is violated when the trust relationships given in the previous section are violated. We assume that tiers cannot learn information by means other than direct communication, i.e. Server cannot obtain the list of purchases through side channels, for instance, unless it is directly shared by Client. Our applications that use secret sharing (Waze and Slice in Section 7) also assume that P2P clients do not collude.

4 Cost Models & Optimizations

This section discusses ZØ's cost modeling approach to optimizing zero-knowledge computations. As outlined in Section 3, in many cases one zero-knowledge engine will outperform the other on a particular computation by a significant factor, giving ZØ a key opportunity to optimize the code it produces. ZØ optimizes zero-knowledge regions by building detailed performance models that characterize the cost of building and verifying zero-knowledge proofs in each engine. We are able to accomplish this with reasonable accuracy because the execution depth of zero-knowledge regions is statically-bounded (a necessary condition imposed by the underlying engines), and the evaluation of zero-knowledge code universally relies on a few primitive operations. This allows ZØ to build static *cost models* as polynomials over the number of primitive operations each region must execute.

Section 4.1 discusses local optimizations within a given zero-knowledge region to decide which back-end to use. Section 4.2 proposes a split for the entire application designed for maximal performance.

4.1 Local Optimization

In order to build cost models for ZQL code, we execute the F# "object code" generated by ZQL's compiler *symbolically*. Symbolic data is represented by polynomials that characterize the size of the corresponding concrete data, or structured sets of polynomials in the case of structured data types. The symbolic operation for each ZQL operation accumulates terms on a polynomial that characterize the cost of that operation in terms of the size of its input data, and returns a new polynomial that characterizes the cost of producing of the result. Because the execution depth of iteration commands is always a polynomial function of the size of the inputs, and ZQL programs do not contain branching, accumulating a cost polynomial by symbolic execution necessarily accounts for *all* of the operations contained in a ZQL program.

Recall that Pinocchio compiles C code into a circuit, which is evaluated by a specialized runtime to produce and verify zero-knowledge proofs. The Pinocchio runtime executes roughly the same code to evaluate every

	ZQL			Pinocchio		
	Setup	**Prover**	**Verif.**	**Keygen**	**Prover**	**Verif.**
FitBit	0.01	1.81	0.10	0.39	0.20	0.00
Waze	0.11	0.29	0.25	0.04	0.02	0.00
Loyalty	0.03	0.35	0.11	0.31	0.20	0.00
Slice	0.06	0.41	0.32	0.05	0.03	0.00
Average	0.05	0.72	0.20	0.20	0.11	0.00

Figure 5: Absolute regression error (in seconds).

circuit, varying only on the number of times each operation is executed to handle every element of each input list and every operation in the circuit. We build a set of static polynomials that characterize the execution time of the runtime in terms of the size of the input circuit, i.e., the number of I/O wires and multiplication gates it contains. For example, the cost of the verification stage is given by the polynomial:

$$ExpMulB \times NInputs + 12 \times Pair + VerifyConst$$

In this polynomial, *ExpMulB* corresponds to the amount of time taken to complete a multi-Exponentiation on the Pinocchio's base elliptic curve, *NInputs* to the number of input wires in the circuit, *Pair* to the field pairing cost [31], and *VerifyConst* to a fixed setup cost for the verification stage. Similar polynomials are derived for the other stages of Pinocchio's runtime.

We use least-squares regression to derive coefficients for all models except those for Pinocchio's compute-stage model, which contains a non-linear term corresponding to the $O(n \cdot \log^2 n)$ runtime of polynomial interpolation. To cope with the non-linearity in Pinocchio's compute-stage model, we use the Gauss-Newton method [33] with at most 1,000 iterations and a randomly-chosen starting point.

Cost-fitting results: To derive the necessary coefficients for our models, we built a regression training application in ZØ consisting of several basic operations likely to appear in zero-knowledge applications. The training application takes as input a list of integers, and computes an aggregate sum, scalar product, second-degree polynomial, boolean mapping, and table lookup on the list. We compiled this application to use both all-ZQL and all-Pinocchio zero knowledge computations, and ran it ten times for each zero-knowledge engine using a fixed list size ($n = 100$). We performed regression to learn coefficients corresponding to the execution time of each primitive operation appearing in the cost model. We then compiled a representative subset of the applications described in Section 7 to use either all-ZQL or all-Pinocchio zero-knowledge computations, executed each zero-knowledge region ten times, and recorded the deviation between execution time predicted by the regression-trained cost models and the mean execution time observed over all experiments for a given application. Figure 5 presents the prediction error of the trained cost models in terms of the total zero-knowledge execution time in seconds. Note that the models derived for Pinocchio are generally more accurate in terms of relative error than those for ZQL, but the error in both cases is quite small: the greatest Pinocchio error is 0.39 seconds (on FitBit's key generation routine), while the greatest ZQL error is 1.81 seconds (on FitBit's prover routine). The coefficient of determination (R^2) for each performance model is at least 0.98, indicating a precise fit of the models to the execution time.

Summary: To summarize, ZØ is able to build performance models of zero-knowledge regions that predict actual execution time within tenths of a second in most cases, which provides ample accuracy to make a correct decision when selecting zero-knowledge engines at compile-time.

4.2 Global Optimization

ZØ builds cost polynomials to characterize the expense of each zero-knowledge operation in the target application. However, selecting the least expensive engine for each operation is oftentimes not as straightforward as evaluating each polynomial at a target input size and choosing the engine corresponding to the lesser value — it may be the case that a less expensive operation on the prover's side requires a more expensive operation on the verifier's side, and depending on the application computation may be more expensive for the verifier. Alternatively, there may be several ways to partition an application between tiers while preserving the privacy of variables at each tier, with each partition yielding a different trade-off between computation and communication cost. To address these concerns, ZØ performs *global optimization* on the application to balance the cost of computation and communication among differentiated tiers.

Performance of global optimization: We implemented our global optimization algorithm as part of the ZØ compiler. We use CCI2 to traverse the AST of the target code, and our cost modeler to generate the objective function.

	Constr.	**Time**
FitBit	179	1.50
Loyalty	38	0.01
Waze	263	2.65
Sice	230	2.14

Figure 6: Global optimization performance, showing solver time in seconds for the benchmarks in Section 7.

To perform the constrained optimization needed to find an optimal solution, we used the Nelder-Mead method [33] with at most 100 iterations. We looked for integer solutions over the full space of tier splittings.

The results are presented in Figure 6. Each application resulted in between 30 and 300 constraints, and the constraint solver found an optimal solution in under three seconds for all applications. Because Nelder-Mead is an

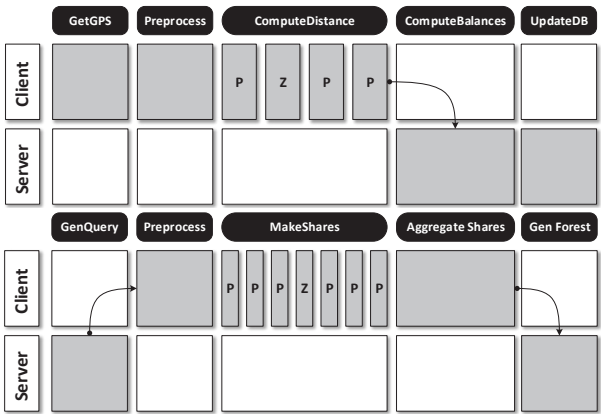


Figure 7: Splits produced by global ZØ optimizations, for FitBit and Slice. For each phase of the computation, grey cells indicate computation location (or tier) chosen by the optimizer, with **P** and **Z** denoting ZQL and Pinocchio back-ends, respectively.

approximate numerical optimization algorithm, it is possible that it would return a *local* minimum.

However, we checked the solution returned for each application, and verified that it corresponded to the true global minimum. Figure 7 shows examples of ZØ-computed global splits for two representative applications.

5 Implementation

In order to make privacy analysis, zero-knowledge translation, and aggressive optimization feasible for the programmer, ZØ supports a subset of C# that includes certain LINQ (language integrated queries [34]) functionality and support for external code. To ensure that the external code does not interfere with the privacy, integrity, and optimization goals of ZØ, the contexts in which it is allowed are limited in some cases. The syntax accepted by ZØ is summarized in Figure 8.

The main program is structured into three parts: an initialization routine (InitBlock, contained in a method `Initialize`), the main body (MainBlock, contained in a method `DoWork`), and the worker methods (MethodDef). The initialization routine may consist of a sequence of arbitrary C# assignment statements, including calls to methods in external libraries not written in ZØ's input language. The main block consists of a sequence of method calls, assignment statements, and sleep statements. Each method call in the main body must be to a worker method defined in the ZØ application.

Zero-knowledge regions: The body of each worker method can contain calls to external methods, standard C# arithmetic and Boolean operations, and a subset of the standard LINQ data processing operations. Regions comprised of LINQ operations can be converted into zero-knowledge proof-generating object code using either available zero-knowledge engine (ZQL or Pinocchio). The supported LINQ operations include Select, Aggregate, First, and Zip. Select provides the ability to project the data in one list into a new list, while performing arithmetic and Boolean operations on each item in the original source list. Aggregate provides the ability to compute iterated functions over a list, maintaining an order-sensitive state through the iteration, which is eventually returned as the result of the operation. First provides the ability to perform searches over lists, using a programmer-defined predicate to determine which element of the list to match. Finally, Zip provides the ability to combine multiple lists, applying arithmetic and Boolean operations to each pair of items from the original source lists.

Zero-knowledge regions are specified by the programmer using a pair of methods ZeroKnowledgeBegin and ZeroKnowledgeEnd. Because zero-knowledge computations provide both integrity and privacy, these annotations serve a dual purpose. First, the programmer is denoting that the variables which are *live* [1] at the end of a zero-knowledge region are trusted across all tiers: the values have accompanying proofs that any tier can examine to verify that the computations in the zero-knowledge

Main program definition

Program	::=	InitBlock MainBlock MethodDef* TypeDef*
InitBlock	::=	CSMethodSig VarDecl*
MainBlock	::=	CSMethodSig WorkerStmt+
MethodDef	::=	CSMethodSig (ExternCall | LinqStmt)+
TypeDef	::=	`class` Id { CSFieldDef + }
CSMethodSig	::=	PrivacyAnnot CSType Id(. . .){ . . . }

Statements

WorkerStmt	::=	SleepStmt | CallStmt | ZKAnnot
SleepStmt	::=	`WorkerSleep`(Integer, Integer, Integer)
CallStmt	::=	(Id =)? MethodCall
ExternCall	::=	`return External.`Id"("Id*")"
LinqStmt	::=	(Id =)? LinqExpr
VarDecl	::=	(PrivacyAnnot | SizeAnnot)? Id(= CSExpr)?

Expressions

Lambda	::=	"("Id*")" ⇒ LambdaExpr
LambdaExpr	::=	MethodCall | ArithOrBoolExpr | FieldExpr | NewObj
LinqExpr	::=	LambdaLinqExpr | ZipLinqExpr
LambdaLinqExpr	::=	Id.LambdaLinqId(Lambda)
LambdaLinqId	::=	Select | Aggregate | First
ZipLinqExpr	::=	Id.`Zip`(Id, NewAnonObj)
MethodCall	::=	Id "("LambdaExpr*")"
NewObj	::=	NewAnonObj | NewStaticObj
NewAnonObj	::=	`new` {(Id = LambdaExpr)+}
NewStaticObj	::=	`new` MethodCall
FieldExpr	::=	Id.`fld`⟨Type⟩(Int)

Annotations

ZKAnnotat	::=	ZeroKnowledgeBegin() | ZeroKnowledgeEnd()
PrivacyAnnot	::=	[Private(T_L)]
SizeAnnot	::=	[MaximumInputSize(Int+)]

Figure 8: BNF syntax for the subset of C# supported by ZØ. Entities prefixed with CS correspond to the corresponding C# syntax entity.

region are performed correctly. Second, these regions serve to *declassify* private values that are used as inputs to a zero-knowledge region; this is in line with the approach taken by ZQL [16]. Because the inputs to zero-knowledge regions are kept private, except in cases where the computations are in some way invertible, the output values that depend on these inputs are considered public to all tiers.

Formal reasoning about composing proofs obtained from different zero-knowledge back-ends remains an avenue for future work. Because this work involves experimentation with very recent cryptographic tools, we are not aware of a readily-available composition theorem that would support reasoning about Pinocchio and ZQL.

Code splitting: ZØ partitions the given target application into code that runs on multiple tiers, inserting marshalling and synchronization code [20, 24] as necessary to ensure that the compiled functionality matches that specified in the original input program. The rewrite process is implemented as a bytecode-to-bytecode transformation within the CCI 2 rewriting framework for .NET [27]. We assume that the target tier for each method is provided as input to the compiler by the optimizer, as described in Section 4.2.

Code partitioning between tiers takes place at method granularity, and data partitioning is determined by the chosen code partition; data is transmitted between tiers on-demand, with all of the data represented by a variable used by a particular method being transmitted at once as it becomes available. Only worker methods can be split between different tiers, so all external code referenced by the application is present on each tier. This allows the compiler to avoid a potentially expensive deep-dependency analysis of the referenced external code, while keeping the dependency analysis of the target application localized to `DoWork`.

Runtime support: The architectural principle that guides ZØ's tier-splitting algorithm can be summarized as follows: *whenever possible, delegate the data communication and synchronization operations necessary to support functionality to a runtime API.* Each application compiled by ZØ is linked to a runtime library that provides an API for communicating data and synchronization between separate tiers. When the compiler performs tier splitting, rather than inlining complex code to perform the tasks, simple calls to this API are inserted to perform the "heavy lifting" of tier crossings at runtime.

6 Translating LINQ to Zero-Knowledge

Our compiler translates specified statements containing LinqExpr components in the worker methods into code that generates zero-knowledge proofs of knowledge. To accomplish this, ZØ relies on two *zero-knowledge back-ends*: ZQL [16] and Pinocchio [31]. Each back-end is itself a compiler, accepting as input an expression of a computation, and producing executable code to produce a zero-knowledge proof of the computation for a given set of inputs. As such, each back-end supports its own *expression language* with significantly different characteristics. The challenge addressed in this section is the translation of the common subset of LINQ supported by ZØ into the expression languages of these back-ends.

Figure 1 in the gives an overview of our back-end compilation process for ZQL and Pinocchio. The details differ widely for each back-end, converging only on the first and last steps which correspond to lifting low-level intermediate language code into a higher representation and inserting I/O marshaling instructions before and after the compiled object code. This divergence of functionality is necessary given the differences between the two expression languages: ZQL's expression language is essentially a small subset of pure standard ML, whereas Pinocchio's is a subset of C with restrictions on data types and loop bounds. Because the subset of LINQ functions supported by ZØ corresponds to a small core of functional expressions, translating from ZØ to Pinocchio is much more involved than to ZQL.

6.1 Pinocchio

The structure of C code is substantially different from the types of LINQ queries allowed by ZØ, and Pinocchio's additional restrictions make translation more complicated yet. First, all list sizes used in the Pinocchio expression must be statically-declared, and any operation over a list requires a static value to bound the corresponding loop statement. The LINQ commands in ZØ do not have these restrictions, so we must find a way to derive the needed information. Second, many expression forms in ZØ's LINQ commands have no corresponding expression form in C: they must be converted into statements whose side-effects are available as sub-expressions to enclosing expressions.

To perform translation to Pinocchio, ZØ follows a three-step process. First, static values for the size of each identifier that refers to a list value are derived using a constraint solver. The basis for this computation is a set of annotations provided by the developer, which indicate upper bounds on the sizes of certain input lists.

List Size Resolution: As previously discussed, Pinocchio requires static sizes for all lists and list operations, so our translation procedure requires a mapping from identifiers (for those that refer to list objects) to size constants. To produce such a mapping, we use a constraint resolution procedure over a set of bounding constraints generated by traversing the source expression. The rules for generating the constraints are given in Figure 9. Each rule is of the form $\Gamma, \textit{Syntactic Element} \Rightarrow \Gamma'$, where Γ

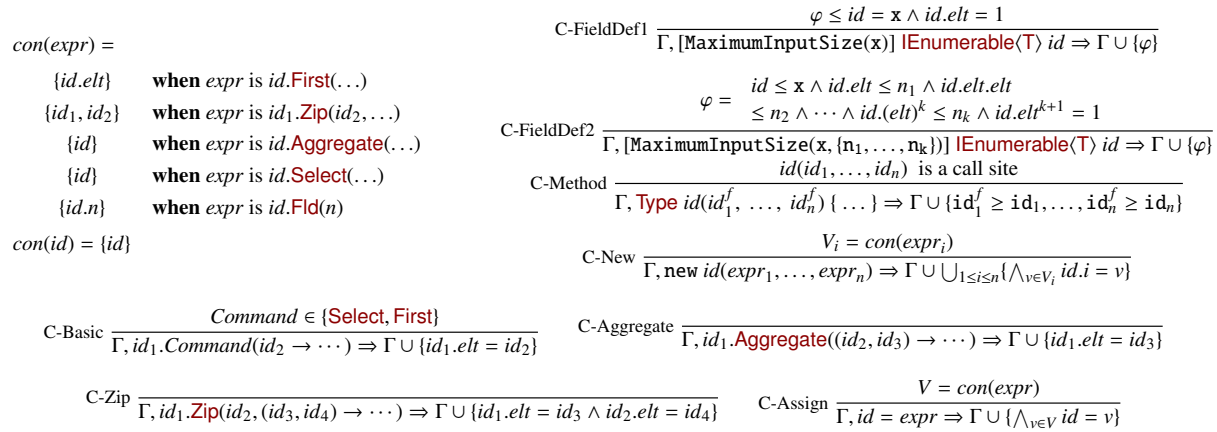


Figure 9: List size constraint generation rules. Γ is a set of constraints.

and Γ′ are sets of constraints. The constraints for each LINQ command are straightforward. The outcome of Select, Aggregate, and Zip operations has the same size as the input variable(s). The outcome of a First statement has the size of the elements contained in the input list.

The rules are invoked by a procedure that traverses each node of the program's AST, and performs syntactic matching on the entity represented by each node and the *Syntactic Element* of each rule. As the traversal proceeds, a list of constraints is maintained, and updated when rules match AST nodes. When the AST traversal completes, the set of constraints generated is passed to Z3 for resolution. If the constraints are satisfiable, Z3 will produce a model that associates constraint variables to integers that satisfy the original constraints. This model contains all of the information needed to derive the needed mapping between identifiers and list sizes.

Type Generation and Function Isolation: Pinocchio requires static sizes on all arrays and loop bounds. To accomplish this, ZØ creates a new `struct` type for each list with a distinct base type and size in the original program. Each new type has two fields: a static array and a constant defining the size.

Once types for each identifier are established, each sub-expression in the source statement is converted to a function body. To see the need for this step, consider the statement x.Select($el \rightarrow el$.Select(...)). C has no expression form for the functionality needed by the Select command, so both expressions must be converted into loop statements. Rather than placing the loop statements in the same method body and carefully managing side effects and sequencing with other sub-expressions, we isolate the emitted code for the inner Select in a separate function, and emit a call to the new function in its place in the context of the outer Select expression.

The statements generated for each LINQ command are straightforward translations of their defined behavior into basic C; in general, the input loop is iterated over, and the lambda passed to the command is invoked over each element. Field lookups, new object construction, and function calls are rewritten to their C equivalents.

6.2 ZQL

Recall that we only attempt to convert LinqStmt statements into zero-knowledge, so there are four primary functions to convert, in addition to a few additional expression forms. By no coincidence, the four primary LINQ functions correspond closely to the operations supported by ZQL. Figure 2 in the gives a set of rewrite rules that can be used to translate a LinqExpr to ZQL's expression language. Select, Aggregate, Zip, and First calls are translated to map, fold, map2, and find expressions. Lambda definitions and functions calls are translated compositionally, by first translating sub-expressions and then building a new construct in the target language. Object creation using new is translated into tuple construction. Recall that user-defined types in a ZØ program must expose a single constructor that assigns all fields of the type; field names are translated into a tuple order using the constructor signature. Similarly, field accesses using fld are translated into a let binding that returns the appropriate tuple component; the translation consults the target identifier's type constructor to deduce the number of fields in the type.

7 Motivating Case Studies

This section presents six case studies in ZØ, that are the focus of our experiments in Section 8. Similarly to [16], we assume that the sensor readings devices can are trusted and untampered with, and come signed by their producer, but the machine or mobile phone (Client tier) that performs the distance computation is not.

1) Walk for Charity with FitBit: Several programs exist for paying users for the amount of physical exercise they perform, either directly in the form of rewards, or indirectly by making charitable donations on their behalf,

such as `earndit.com`. This works by requiring users to log their exercise habits using a FitBit or other sensor device to measure the distance the user walks, runs, or bikes, and send the logs to a centralized server.

Privacy: The user may not want to reveal their *detailed physical activities* or *exercise route* to a relatively untrusted third party.

Integrity: The service is spending money on the basis of distance derived from sensor logs. If the distance computation can be subverted, the possibility for fraud arises, analogously to pay as you drive insurance [4, 38, 41].

Solution: Keep all sensor readings local to the user's machine (laptop or mobile device), perform the distance computation locally, on the client, send the result of the distance computation to the centralized third-party server. Use ZKPK to ensure that the distance computation is performed correctly. This approach is similar to what has been advocated for smart metering [35].

2) Supervised Studies in Social Sciences: Many scientific studies, especially in medical and social sciences, require subjects to wear sensors and undergo protocols that provide information about their physiological and psychological state. A study that seeks to understand the effect of common workplace events on worker's stress levels might require a participant to wear a galvanic skin response sensor and a camera to detect face-to-face interactions.

Privacy: Participants may have concerns about the use of their physiological measurements or, most prominently, the processing of images taken from their cameras.

Integrity: These studies typically involve payment given to subjects. Subjects concerned about their privacy, or those who simply do not want to wear intrusive sensor devices, have an incentive to *fake* their data.

Solution: Have all sensors associated with the study report readings to the subject's machine (desktop or mobile phone). This machine performs aggregate computations relevant to the actual study on the readings, reporting results and discarding the raw sensor readings. ZKPK is used to ensure that the readings are processed correctly.

3) Personalized Loyalty Cards: Many of today's large retailers such as Target, BestBuy, etc. use customer loyalty cards to encourage repeat visits. Typically, the customer must enroll in a loyalty program, and receive a card that can be applied to receive discounts in future visits. Recently, certain retailers (e.g., Safeway) have begun personalizing this process by using the customer's past purchase history (available because of the association between checkout and loyalty card) to create discounts available only to one particular customer. Depending on the retailer, these discounts can be sent to the customer's mobile phone, or applied automatically at checkout.

Privacy: Many people are not comfortable with a retailer tracking their purchases. This is most readily illustrated by a recent scandal with Target discovering that a teenage girl was pregnant before her parents did [14].

Integrity: Retailers offer discounts on the basis of past purchase history. If a customer could fake a purchase history, they might be able to obtain a discount for an item of their choosing. Moreover, having a reproducible strategy for "generating" discounts might create a serious problem for the retailer, similar to those experienced by some retailers that were overly generous in offering Groupons [32].

Solution: The solution is discussed in Section 2.1.

4) Crowd-sourced Traffic Statistics: Several mobile applications such as Waze (`waze.com`) and Google Maps provide traffic congestion information to end-users based on the combined GPS readings of the users.

Privacy: Users do not want to share their location with the app's servers, or the general public (in the case of a distributed protocol).

Integrity: The app needs reliable GPS readings from users to provide its core functionality. If users wish to "game" the system by providing fake GPS readings while receiving the end-product, the integrity of traffic data is compromised for everyone.

Solution: Let the users keep their GPS readings local, and take part in a distributed protocol to compute local density information for transmission to the app's central server. Clients represent their location on a map using a vector, represented as a set of secret shares, which can be added to the other clients' vector shares to derive the overall traffic density map. When each client sends their summed shares to the server, it can reconstruct the density map by combining the shares, as detailed in the appendix.

5) CNIDS: Collaborative intrusion detection (CNIDS) has long been a goal of security practitioners [25]. In the CNIDS scenario, multiple (distrustful) organizations share the results of their network intrusion detection sensors, to provide their peers with advanced warning about possible threats. A practical approach involves sharing IP blacklists: when an IP generates a valid NIDS alert on one organization's network, the IP is recorded and sent to the other participating organizations.

Privacy: NIDS operate on highly sensitive data — raw network traces. Organizations participating in CNIDS do not want to share their traces with other organizations, and in many cases, may be prohibited from doing so by law or organizational policy.

Integrity: Given the privacy concern and the benefits of participating, some organizations may want to freeload by suppressing their own NIDS alerts. Additionally, if

an adversary manages to compromise a participating network, it may choose to suppress or even generate false alerts, which may result in a denial of service for the targeted IP address.

Solution: Provide a ZKPK for the NIDS signature-matching process, to prove that a claimed intrusion is correct according to the signature. Note that this approach assumes that raw network data coming into the NIDS has not been tampered with, but that the machine performing the signature matching may not be trusted.

6) Slice: Organizing Shopping: Slice (`slice.com`) is a service that takes as input a user's past purchase history from their email mailbox, and provides various services using that data. One such service is product recommendation — given everybody's past purchase history, slice can build classifiers that predict a likely "next" purchase.

Privacy: Handing one's entire purchase history to a profit-driven third party has obvious privacy implications. So does the troubling need to share one's email credentials with Slice at the moment.

Integrity: A user, particularly one concerned about privacy, might provide fake data to Slice in order to obtain the useful classifier, which would pollute Slice's data for everyone and jeapordize Slice's ability to profit from the classifier.

Solution: Keep the user's purchase history local, and have the users take part in a distributed protocol in order to produce the classifier for Slice. Use ZKPK to ensure that no user is able to subvert the distributed classifier computation.

8 Experimental Evaluation

All experiments were performed on a Windows Server 2012 R2 machine with two 3.0 GHz 64-bit cores with 8 GB of RAM. All reported timing measurements correspond only to the zero-knowledge portion of the application's execution time, as this is the only portion that our compiler attempts to optimize.

The execution time of the ZK code is generally much higher that of the rest of the application, so focusing on these parts gives an accurate picture of the overall execution time. Each zero-knowledge proof generation and verification task was terminated after ten minutes. Our implementation uses 1,024-bit RSA keys for ZQL computations. Integers in Pinocchio circuits were configured to have 32-bits for comparison operations, and operate over a 245-bit field.

Figure 11 summarizes the key performance results from our experiments. We found that the ZØ-generated code gave significant performance benefits both in terms of computation time and proof size: up to 40× runtime speedup, with most proofs below 1 MB (the largest being ≈ 1.9 MB). Furthermore, we saw that global optimization is necessary to arrive at an ideal performance profile: some applications perform noticeably worse at one tier, but in each case the speedup at another tier was always greater. For example, the code ZØ generated for the Waze server ran ≈ 2× slower than Pinocchio's on average, but latency on the client tier was reduced ≈ 4×.

Scaling	ZØ scales to all application configurations. Others may time out or fail to compile in fewer than 20 minutes on some parameter settings: 100-byte traces (NIDS), >100 peers (Slice), large automata (Loyalty).
Latency	ZØ improves up to 40×, ≈ 5–13× on average
Proof size	ZØ almost always less than 1 MB, at most 1.5 MB. ZQL proofs can be tens or hundreds of MBs.
Global tradeoffs	ZØ may be slower at one tier (2× slower for Waze server), but savings at other tiers is always much greater (4× faster for Waze clients)

Figure 11: Performance summary.

Figure 12 shows the latency speedups across all applications. The average speedup delivered by ZØ is 3.3× compared to Pinocchio and 7.4× compared to ZQL.

Results: Space limitations do not allow us to present our measurements exhaustively. Instead, Figure 10 shows a sample of the runtime characteristics for our target applications. Rather than giving raw execution times, the results are broken into three categories: *throughput*, *latency*, and *proof size*. These metrics were selected to more clearly depict the impact of zero-knowledge techniques on each application.

Throughput: Figure 10(a)–(c) shows the results of three experiments involving throughput. Figure 10(a) shows the server's throughput for the Waze application, which corresponds to the number location updates per minute the server can handle as the number of users (n) increases. Notice that Pinocchio outpaces both the hybrid and ZQL compilations by about 2× on average. This is a result of the global optimization engine: verification in Pinocchio is very fast, whereas the time to construct a proof can be quite slow: in this case, the proof construction phase was up to 7× slower than the hybrid solution. This is critical, *as proof construction takes place on the client where resources are especially constrained for the application.* The discrepancy in resources is correctly used by ZØ to optimize for a lighter client workload at the expense of greater server overhead.

Figure 10(b) shows the number of random forest construction queries per minute the Slice server is able to handle, as the number of participating peers increases. As with Waze the Pinocchio solution dominates the ZØ solution at all data points because of the greater expensive of constructing proofs on the client, where the Pinocchio solution is up to 4× slower than ZØ.

Figure 10(c) shows the number of intrusion alerts per minute the collaborative NIDS server can handle as the number of bytes in the intrusion trace increases. Notice

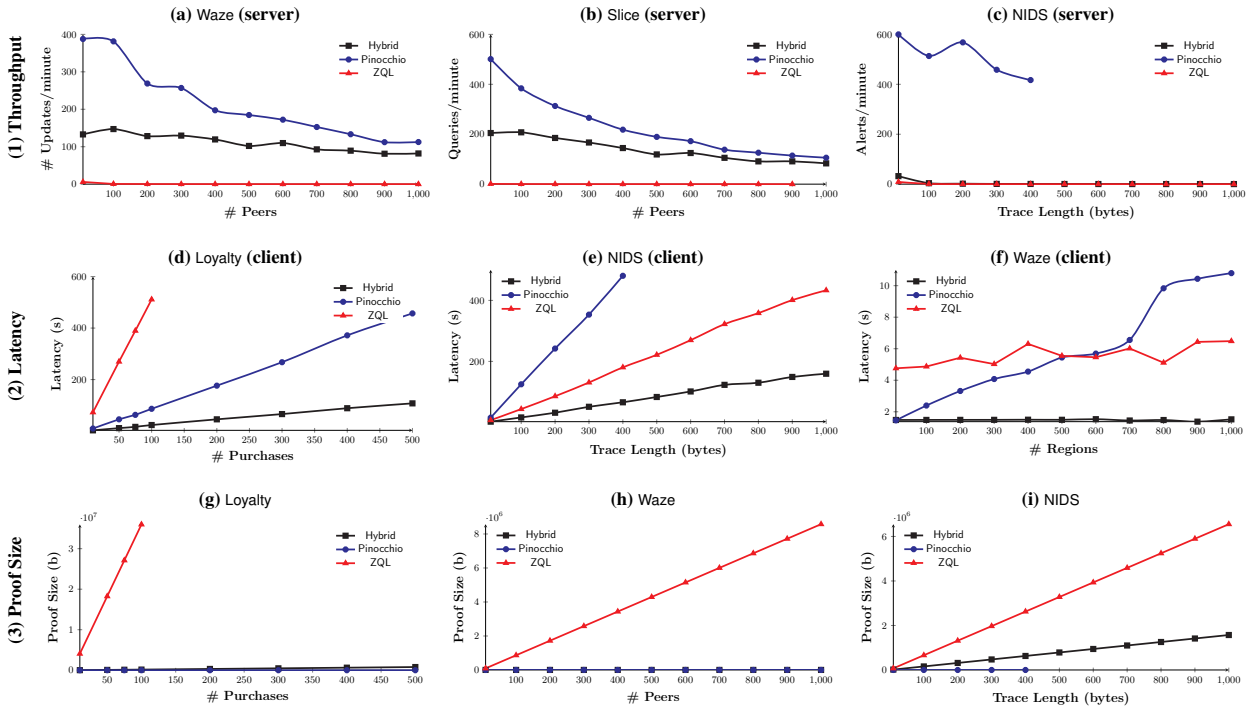


Figure 10: (1) Throughput, (2) latency, and (3) proof size for a characteristic sample of application functionality.

that Pinocchio outperforms at a few small data points, but fails to scale to any larger points. This is not because the server-side component is unable to scale, but rather the client timed out at these settings. For the remaining points, the ZØ solution outperforms the others by about 4×, and is the only solution that is able to scale to even the modest intrusion trace length of 1 KB.

Latency: Figure 10(d)–(f) shows the results of three experiments involving latency. Latency is always measured in seconds, and has a uniform upper bound of 600 seconds, which corresponds to our experimental timeout.

Figure 10(d) shows the latency of the client side of the Loyalty application as the number of purchases used to personalize discounts (n) increases. The ZØ solution far outpaces both alternatives at all data points (4–22× improvement). These experiments were performed for an automaton with about 75 edges. We found that when we scaled the automaton to more realistic sizes (a few thousand edges), the ZØ solution was the only one capable of completing *any* number of purchases before timing out, and the Pinocchio compiler timed out after 20 minutes. For longer purchase histories, the ZØ solution completes in just over 1.5 minutes, which is ample time if the application is location-aware and begins proving a set of discounts when the user enters the store.

	Pinocchio		ZQL	
	Mean	Max	Mean	Max
FitBit	6.4	6.6	4.5	4.7
Study	1.0	1.0	39.7	40.3
Loyalty	4.1	4.2	10.1	21.8
Waze	4.0	7.1	4.3	4.7
CNIDS	5.3	7.3	2.7	2.7
Slice	2.5	4.1	8.1	12.9
Mean	**3.3**		**7.4**	

Figure 12: Latency speedup factors for each application; averages use geometric mean for proportional speedup.

Figure 10(e) shows the NIDS client's latency to demonstrate that a single intrusion is present in a trace. Pinocchio times out at all points beyond 300 bytes, whereas ZØ is about 2.7× faster than ZQL. Otherwise, we see that as long as intrusions are spaced more than two-and-a-half minutes (159 seconds) apart, the NIDS client has enough time to build proofs for each intrusion trace.

Figure 10(f) shows the latency of the Waze client to send traffic statistics for a single location query as the size of the map (n) increases. First notice that the ZØ solution is essentially constant, not varying by more than 1.5 seconds between any two data points. The other solutions require as much as 4–7× as long to process a query on the client, which will limit the quality (i.e., recency) of the statistics the server is able to gather over time. Second, notice that at about $n = 700$, ZQL becomes more performant than Pinocchio. This is because as the map increases, the size of the lookup table needed to encode the regions increases. Pinocchio is not able to perform lookups as quickly as ZQL, so the portion of the computation needed for lookups becomes more significant at higher values of n. ZQL performs worse at lower values because most of the computation corresponds to the multiplications needed to compute secret

shares, which it does not complete as quickly as Pinocchio.

Proof Size: Figure 10(g)–(i) shows the results of experiments involving the size of the zero-knowledge proof in various applications. We always measure in bytes, and do not display a curve for the Pinocchio solutions, as it is constant across input size and is usually too small to distinguish on the same scale as the ZQL and ZØ solutions. Figure 10(g) shows the proof size for the Loyalty application as the number of past purchases (n) varies. While the Pinocchio solution of course dominates the others by this metric (864 bytes), as we know from previous experiments (Figure 10(d)) it does not scale in terms of Latency. The ZØ proof size remains nearly constant, always under 500 KB, whereas the ZQL solution requires at least three megabytes (to perform the inequality checks at the beginning), and finishes at about 100 megabytes. Note that we obtained the point at $n = 300$ despite the timeout, by letting the prover run for longer in this single instance. Because the Loyalty application needs to communicate this proof wirelessly to a POS terminal, size is crucial, and the ZØ solution offers the best overall characteristics in terms of size and latency.

Figure 10(h) shows the proof size for the Waze application as the number of peers varies. Again, Pinocchio dominates (2 KB), but the tradeoff in latency for this proof size is quite high (Figure 10(f)). The ZØ proof size remains constant at around 5 KB because the only processing done by ZQL is table lookups, which have a constant proof size. The ZQL solution requires 20 megabytes for 2,500 clients, and 8 megabytes for 1,000 clients, making it untenable given that the clients need to transmit proofs frequently over cellular networks.

Figure 10(i) shows the proof size for the NIDS application as the intrusion trace length increases. The Pinocchio proof is about 1 KB, but again the tradeoff in latency makes this characteristic mostly irrelevant. The sizes for the ZØ and ZQL solutions are both linear, with the ZØ solution offering a savings of about 4× at all data points. This is a significant savings, considering that false positives may be frequent, so the client may need to send proofs to the server almost continuously.

9 Limitations and Future Work

Proof of security: The main piece of outstanding work for ZØ is a formal argument of security. Because ZØ composes non-interactive zero-knowledge proofs from distinct back-ends, the security guarantees given by the original back-ends do not necessarily readily translate to the final optimized code produced by the compiler. In future work, we hope to characterize a unified threat model that encompasses those of both back-ends, as well as a composition theorem that demonstrates the safety of ZØ's modular compilation philosophy.

Optimization robustness: One concern is that a developer may unwittingly write code in a zero-knowledge block that ZØ compiles into very inefficient code. In general, ZØ's cost models should allow it to select the best back-end most of the time. In certain close cases, where the performance difference between back-ends is slight, discrepancies between ZØ's model coefficients and the characteristics of the target architecture may lead it to select the less-efficient back-end. However, as the difference between back-ends is small to begin with in such cases, the absolute performance penalty will likely be small as well.

As non-interactive zero-knowledge is still significantly more expensive than "normal" computation even in the best cases, the programmer must be careful not to place unnecessary statements inside of a zero-knowledge block. Additionally, if the programmer places inaccurate size annotations on data structures, i.e., annotations that are significantly larger than the average workloads encountered in practice, then the cost models used by ZØ during optimization might not characterize the actual performance requirements of the application; this can lead to sub-optimal performance.

Hardware integrity: Many of the applications discussed in this paper gather data from trusted hardware devices. The zero-knowledge facilities in ZØ ensure that the results of computations performed on such data can also be trusted, i.e., they were derived by the code originally intended by the application developer. However, zero-knowledge proofs might not provide all of the guarantees needed to realize an intended high-level security goal in some cases.

For example, nothing prevents a malicious user from "fooling" the FitBit application by physically manipulating the hardware to register more steps than were actually taken. In these cases, ZØ increases security by ensuring that attacks on the application code will not succeed, so that more-expensive hardware-layer attacks are necessary. Whether this makes an attack on a given application sufficiently difficult, or economically infeasible, is a point to be carefully considered as part of an end-to-end security strategy.

10 Related Work

Tier-Splitting and Language Methods: A number of compilers exist that enable automated tier-splitting in some form. In the context of web programming, Google Web Toolkit (GWT) [20], Volta [24], Links [11], and Hilda [43] are among the pioneering efforts. ZØ is closest to Volta and GWT, allowing developers to supply a single piece of code that is compiled into separate modules for the client and server. Unlike those projects, ZØ uses cost models of execution time and data size to derive an optimization problem whose solution represents

an ideal division of functionality between tiers.

Others have used tier splitting to provide security and privacy guarantees. SWIFT [10] builds on the JIF [28] language, incorporating security types for confidentiality and tier-splitting for web applications. To accomplish this, information flow constraints are embodied in an integer programming problem whose solution corresponds to a valid (e.g., secure) placement of code onto tiers that minimizes the number of messages that must be transferred. Unlike ZØ, SWIFT does not explicitly account for data size and transfer time when looking for a split that is likely to maximize performance.

References	TS	P	I	IL	O
[11, 20, 24, 43]	✓				
[10]	✓	✓			
[3]	✓	✓	✓		
[40]	✓		✓		
[2, 3, 16, 26, 31]		✓	✓		
[16]	✓	✓	✓		
[31]		✓	✓		✓
ZØ	✓	✓	✓	✓	✓

Figure 13: Comparison of distributed and secure compiler efforts. **TS** = Automatic tier-splitting; **P** = Privacy enforcement; **I** = Integrity enforcement; **IL** = Integration with widely-used languages and runtimes; **O** = Optimizing code generation.

Backes *et al.* [3] presented a compiler for distributed authorization policies written in Evidential DKAL [6], an authorization logic that supports signature-based proofs. The use of zero-knowledge proofs allows principals to prove access rights based on sensitive data without directly revealing its content. ZØ differs in its applicability: ZØ allows developers to use C# as part of a larger .NET application, whereas this work translates authorization logic formulas into cryptographic code.

Others have addressed the problem of untrusted client-side computation in various contexts [21, 22, 40, 42]. A similar notion of integrity was presented in Ripley [40], which prevents client-side cheating in web applications by efficiently replicating client-side computations on the server. Unlike ZØ, Ripley's mechanism does not preserve privacy.

Zero-Knowledge Proofs: Zero-Knowledge proofs of knowledge [5] have been extensively studied. Schemes have been developed for various types of relations and computations [7, 8, 19, 36]. Several projects have sought to provide zero-knowledge compilers [2, 3, 16, 26, 31] that take a proof goal and produce executable zero-knowledge code. The first set of zero-knowledge compilers [2, 3, 26] required specifications of cryptographic protocols [9], and so are difficult for non-cryptographers to use. The second generation [16, 31] are geared towards generating ZK code for general computations expressed in restricted high-level languages. Our work makes extensive use of these compilers to optimize zero-knowledge computation. There are a number of larger projects that incorporate zero-knowledge proofs in order to manage integrity without sacrificing privacy. Applications include privacy-preserving smart metering [35], random forest and hidden Markov model classification [12], and privacy-preserving automotive toll charges [4].

11 Conclusions

This paper paves the way for using zero-knowledge techniques for day-to-day programming. We have described the design and implementation of ZØ, a distributing zero-knowledge compiler which produces distributed applications that rely on ZKPK to provide simultaneous guarantees for privacy and integrity. We build on recent developments in zero-knowledge cryptographic techniques, exposing to the developer the ability to take advantage of these advances without requiring domain-specific knowledge or learning a new specialized language. Most of the heavy lifting is done by the compiler, including cost modeling to decide which zero-knowledge back-end to use and how to split the application for optimal performance, together with the actual code splitting.

Our cost-fitting models provide an excellent match with the observed performance, with R^2 scores at least and .98. Our global application optimizer is fast, completing in under 3 seconds on all programs. Our manual and experimental examination of program splits and back-end choices proposed by ZØ confirms that they are indeed optimal. Using six applications based on real-life commercial products, we show how ZØ makes it viable to use zero-knowledge technology. We observe performance improvements of over 40×. Perhaps most importantly, ZØ allowed many of the applications to scale to large data sizes with thousands of users while remaining practical in terms of computation time and data size. This means that applications which were not feasible using state-of-the-art zero-knowledge tools are now practical in realistic settings.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. We thank the ZQL and Pinocchio developers for graciously providing code and support for our effort.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In *Proceedings of the European Conference on Research in Computer Security*, 2010.

[3] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[4] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Pretp: privacy-preserving electronic toll pricing. In *Proceedings of the Usenix Security Conference*, 2010.

[5] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *Proceedings of the International Cryptology Conference on Advances in Cryptology*, 1993.

[6] A. Blass, Y. Gurevich, M. Moskal, and I. Neeman. Evidential authorization*. In S. Nanz, editor, *The Future of Software Engineering*. 2011.

[7] S. Brands. Rapid demonstration of linear relations connected by boolean operators. In *Proceedings of the International Conference on Theory and Application of Cryptographic Techniques*, 1997.

[8] J. Camenisch, R. Chaabouni, and A. Shelat. Efficient protocols for set membership and range proofs. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, 2008.

[9] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *Proceedings of the International Cryptology Conference on Advances in Cryptology*, 1997.

[10] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure Web applications via automatic partitioning. *SIGOPS Operating Systems Review*, 41(6), 2007.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*. 2007.

[12] G. Danezis, M. Kohlweiss, B. Livshits, and A. Rial. Private client-side profiling with random forests and hidden Markov models. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, 2012.

[13] D. Davidson, M. Fredrikson, and B. Livshits. MoRePriv: Mobile OS Support for Application Personalization and Privacy (Tech Report). Technical Report MSR-TR-2012-50, Microsoft Research, May 2012.

[14] C. Duhigg. How companies learn your secrets. `http://nyti.ms/SZryP4`, Feb. 2012.

[15] T. Fechner and C. Kray. Attacking location privacy: exploring human strategies. In *Proceedings of the Conference on Ubiquitous Computing*, 2012.

[16] C. Fournet, M. Kohlweiss, and G. Danezis. ZQL: A compiler for privacy-preserving data processing. In *Usenix Security Symposium*, 2013.

[17] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. In *IEEE Symposium on Security and Privacy*, May 2011.

[18] F. D. Garcia, E. R. Verheul, and B. Jacobs. Cell-based roadpricing. In *Proceedings of the European Conference on Public Key Infrastructures, Services, and Applications*, 2012.

[19] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the IACR Eurocrypt Conference*, 2013.

[20] Google Web Toolkit. `http://code.google.com/webtoolkit`.

[21] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley, 2007.

[22] S. Jha, S. Katzenbeisser, and H. Veith. Enforcing semantic integrity on untrusted clients in networked virtual environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[23] F. Kerschbaum. Privacy-preserving computation (position paper). `http://www.fkerschbaum.org/apf12.pdf`, 2012.

[24] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Softtware*, 25(5):53–59, 2008.

[25] M. Marchetti, M. Messori, and M. Colajanni. Peer-to-peer architecture for collaborative intrusion and malware detection on a large scale. In *Proceedings of the International Conference on Information Security*, 2009.

[26] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of the Usenix Conference on Security*, 2010.

[27] Microsoft Research. Common compiler infrastructure. `http://ccimetadata.codeplex.com`, 2012.

[28] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1997.

[29] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[30] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[31] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[32] C. Pontoriero. Is groupon a raw deal for publishers? `http://risnews.edgl.com/retail-trends/Is-Groupon-a-Raw-Deal-for-Retailers-73442`, June 2011.

[33] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes, 3rd edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[34] J. Rattz and A. Freeman. *Pro LINQ: Language Integrated Query in C# 2010*. Apress, 2010.

[35] A. Rial and G. Danezis. Privacy-preserving smart metering. In *Proceedings of the Workshop on Privacy in the Electronic Society*, 2011.

[36] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[37] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2010.

[38] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In P. Ning and T. Yu, editors, *Proceedings of the 2007 ACM Workshop on Privacy in the Electronic Society*, 2007.

[39] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In *Proceedings of the ACM Workshop on Privacy in Electronic Society*, 2007.

[40] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing distributed Web applications through replicated execution. In *Conference on Computer and Communications Security*, 2009.

[41] Wikipedia. Usage-based insurance. `http://en.wikipedia.org/wiki/Usage-based_insurance`, 2013.

[42] J. Yan. Security design in online games. In *Proceedings of the Annual Computer Security Applications Conference*, 1993.

[43] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven Web applications. In *Proceedings of the International Conference on Data Engineering*, 2006.

SDDR: Light-Weight, Secure Mobile Encounters

Matthew Lentz[†] Viktor Erdélyi[‡] Paarijaat Aditya[‡]
Elaine Shi[†] Peter Druschel[‡] Bobby Bhattacharjee[†]

[†]*University of Maryland* [‡]*MPI-SWS*

Abstract

Emerging mobile social apps use short-range radios to discover nearby devices and users. The device discovery protocol used by these apps must be highly energy-efficient since it runs frequently in the background. Also, a good protocol must enable secure communication (both during and after a period of device co-location), preserve user privacy (users must not be tracked by unauthorized third parties), while providing selective linkability (users can recognize friends when strangers cannot) and efficient silent revocation (users can permanently or temporarily cloak themselves from certain friends, unilaterally and without re-keying their entire friend set).

We introduce SDDR (Secure Device Discovery and Recognition), a protocol that provides *secure encounters* and satisfies all of the privacy requirements while remaining highly energy-efficient. We formally prove the correctness of SDDR, present a prototype implementation over Bluetooth, and show how existing frameworks, such as Haggle, can directly use SDDR. Our results show that the SDDR implementation, run continuously over a day, uses only ~10% of the battery capacity of a typical smartphone. This level of energy consumption is four orders of magnitude more efficient than prior cryptographic protocols with proven security, and one order of magnitude more efficient than prior (unproven) protocols designed specifically for energy-constrained devices.

1 Introduction

Mobile social applications discover nearby users and provide services based on user activity (what the user is doing) and context (who and what is nearby). Services provided include notifications when friends are nearby (Foursquare [6], Google Latitude [7]), deals from nearby stores (Foursquare), content sharing with nearby users (FireChat [5], Whisper [15], Haggle [50]), messaging for missed connections (SMILE [43], SmokeScreen [27]), lost and found (Tile [13], StickNFind [12]), sharing payments with nearby users (Venmo [14]). At their lowest layer, these applications all discover nearby devices; many also associate previously linked users to discovered devices and provide communication among presently or previously co-located devices.

Most commercially deployed solutions rely on a trusted cloud service [6, 7], which tracks users' activity and location, so that it can match co-located users and relay information among them. Discovery using a centralized matchmaking service forces users to disclose their whereabouts, perils of which have been extensively noted [16, 19, 24, 48, 52]. Instead of relying on centralized services, an alternate class of discovery protocols make use of local, short-range radio-to-radio communication [1, 9, 27, 50]. The common practice of using static identifier(s) in the discovery process [2] leaks information, since it allows an eavesdropper to track a user's locations and movements. To protect against such tracking, previous work [35–37] has suggested that ephemeral identifiers should be used in place of static ones. Simply replacing static identifiers with strictly random ephemeral identifiers is insufficient: while eliminating tracking, it also prevents friends (or users with prior trust relations) from recognizing each other when nearby.

In this paper, we describe a light-weight, energy-efficient cryptographic protocol for *secure encounters* called SDDR. At a high level, secure encounters provide the following properties: 1) discovering nearby devices, 2) mapping devices to known principals (if possible), and 3) enabling secure communication for encounter peers.

Device discovery and secure encounter SDDR performs a pair-wise exchange of a secret with each nearby device. The shared secret enables encounter peers to communicate securely during and after the encounter, anonymously and without trusting a third party (e.g., sharing related content with event participants).

Selective linkability and revocation Additionally, SDDR enables a user's device to be identifiable by specific other users, while revealing no linkable information to other devices. For instance, friends can agree to recog-

nize each others' devices, while third parties are unable to link and track devices upon repeat encounters. Moreover, users can efficiently and unilaterally revoke or suspend this linkability, for instance based on the current time or location (e.g., discoverable by colleagues only during work hours and on company premises).

Challenges: Energy efficiency and DoS resilience In theory, designing a protocol that satisfies the above functional and security requirements is straightforward. For example, an inefficient strawman scheme can be constructed using existing cryptographic primitives. Pairs of devices can perform a Diffie-Hellman key exchange to establish a shared secret, enabling the users to securely communicate. To support selective linkability, two users can participate in a standard Private Set Intersection (PSI) protocol. A user can allow (or disallow) a peer to recognize them in a future encounter by including (or excluding) a past shared encounter secret from the set.

However, as we will show in Section 6, using a full-fledged PSI protocol is impractical. Because the shared encounter secrets (i.e., elements in the set) are high-entropy values, it is possible to implement a secure PSI protocol through an efficient Bloom filter based construction. Unfortunately, even when using an efficient Bloom filter based PSI scheme, the above strawman scheme—implemented naively—has high energy consumption. Specifically, a naive implementation requires a device to wake up its CPU each time it receives a message from a nearby device, an expensive operation for energy-constrained mobile devices. The protocol would deplete the battery in crowded spaces (e.g., a subway train) where hundreds of devices may be within radio range. Furthermore, an attacker mounting a DoS attack could deplete the victim device's battery by frequently injecting messages to cause unnecessary wake ups.

1.1 Contributions

We designed, implemented, and formally proved the security of SDDR, a light-weight secure encounter protocol suitable for resource-constrained mobile devices. Our reference implementation source code (using Bluetooth 2.1 as the short-range radio) is available at http://www.cs.umd.edu/projects/ebn.

Achieving energy efficiency The main feature of SDDR is its *non-interactiveness*, i.e., the encounter protocol consists of periodic broadcasts of beacon messages, which enable both the key exchange and selective recognition. Because the SDDR protocol is non-interactive, the Bluetooth controller can be initialized so that it responds to discovery requests from peers with a beacon message, while the main CPU remains completely in the *idle state*. A device only needs to wake up its CPU when actively discovering nearby peers.

Our evaluation shows that such a non-interactive protocol allows us to improve the energy efficiency by at least 4 times in comparison with any interactive protocol (even if the interactive protocol performs no work), under a typical setting with 5 *new* devices nearby on average during *every* 60 second discovery interval. Under the same parameters, we show that an otherwise idle device running SDDR over Bluetooth 2.1 will operate for 9.3 *days* on a single charge.

First formal treatment of the problem We are the *first* (to the best of our knowledge) to provide a formal treatment of secure device discovery and recognition. We define a security model that captures the requirements of secure encounters and selective linkability, and prove that our solution is secure under the random oracle model (see Appendix A.3).

Applications over SDDR To demonstrate some of SDDR's capabilities, we have modified the Haggle mobile networking platform to use SDDR, enabling efficient and secure discovery and communication via Bluetooth for all Haggle apps. For demonstration, we have modified the PhotoShare app to enable private photo sharing among friends using SDDR selective linking.

Roadmap The remainder of the paper is organized as follows. We discuss related work in Section 2. Next, we review security requirements, formulate the problem and provide security definitions in Section 3. We present details of the SDDR discovery protocol in Section 4, followed by our reference Bluetooth implementation and evaluation results in Sections 5 and 6, respectively. We discuss the properties and implications of SDDR's encounter model in Section 7. We conclude in Section 8.

2 Related Work

Device discovery protocols Several device discovery protocols have been proposed; however, none simultaneously offer the full functionality and security offered by our SDDR protocol. Since SDDR provides secure device discovery and recognition for a large range of mobile encounter applications, it allows developers to focus on their application logic.

	Unlinkability	Selective Linkability	Efficient Revocability	No Trusted Party	Record Encounters
Bluetooth 4.0	✓	✓		✓	
SMILE	✓			✓	✓
SmokeScreen	✓	✓			✓
SDDR	✓	✓	✓	✓	✓

Table 1: Comparison of related device discovery and recognition protocols in terms of supported properties.

Bluetooth 4.0 (BT4) is the most recent version of the Bluetooth standard, introducing a new low-energy mode [3], as well as support for random MAC addresses to be used in communication. Building on top of the MAC address change support, BT4 adds a form of selective linkability in which paired (trusted) devices can recognize each other across MAC address changes, while remaining unlinkable to all other devices. Since BT4 uses a single shared key for all currently linkable users, it does not allow for efficient revocation of a subset of users. Further, BT4 does not natively support encounters with unlinkable devices.

SMILE [43] is a mobile "missed connections" application, which enables users to contact people they previously met, but for who they don't have contact information. The SMILE protocol creates an identifier and shared key with any set of devices that are within Bluetooth range at a given time. Users can subsequently exchange messages (encrypted with the shared key) anonymously through a cloud-based, untrusted mailbox associated with the identifier. Unlike SDDR, SMILE does not address selective linkability and revocation.

MeetUp [44] is an encounter-based social networking application that argues for (and uses) strong authentication within an encounter. This authentication comes in the form of exchanging signed certificates (from a trusted authority) attesting to a public key and picture of a user. However, we feel that in many applications, users should be unlinkable by default, and should not be required to distribute any identifiable information (e.g., public key, user picture) in an encounter. We discuss authentication in Section 4.4.

SmokeScreen [27], a system that allows friends to share presence while ensuring privacy, also implements a selectively linkable discovery protocol for encounter peers. In SmokeScreen's discovery protocol, devices periodically broadcast two types of messages: *clique signals* and *opaque identifiers*. Clique signals enable private presence sharing among friends, announcing the device's presence to all members of a mutually trusting clique. In comparison with SDDR, SmokeScreen requires a trusted third-party service and achieves slightly weaker security: an adversary can infer that two users belong to the same clique, since all users broadcast the same clique signal during each time epoch. Furthermore, SDDR can handle 35 nearby devices for the same energy as 3 devices in SmokeScreen. Additionally, SDDR supports efficient revocation of linked users, which is not possible with cliques in SmokeScreen.

SlyFi [35] is a link layer protocol for 802.11 networks that obfuscates MAC addresses and other information to prevent tracking by third parties. Unlike SDDR, SlyFi does not address selective linkability or revocation, nor does it negotiate shared keys among co-located devices. SDDR includes a Bluetooth MAC address change protocol similar to SlyFi's to prevent tracking.

Related protocols using Bloom filters Bloom filters [20] are a space-efficient probabilistic data structure for set membership. Bloom filters have been used in many cryptographic protocols [23], including (private) set-intersection and secure indexes. However, none of the protocols address the precise problem and security requirements of SDDR.

Secure indexes are data structures that allow queriers to perform membership tests for a given word in $O(1)$ time if they have knowledge of the associated secret. Secure indexes were first defined and formalized by Goh [33], who provided a practical implementation using Bloom filters. Similar work has focused on privacy-preserving searches over encrypted data [26] and databases [54] using Bloom filters. If applied to device recognition, all protocols would allow adversaries to track users due to the static Bloom filter content.

PrudentExposure [56] allows users to privately discover appropriate services, where the user and service belong to the same domain. To maintain user privacy, PrudentExposure relies on Bloom filters containing time-varying hashes of domain identities for intersecting the requested and available domains.

E-SmallTalker [55] and D-Card [25], which builds on E-SmallTalker, support social networking with nearby strangers (E-SmallTalker) and friends (D-Card). BCE [31] enables users to estimate the set of common friends with other users. These protocols would be insecure when applied to the device recognition problem, as none of the protocols use time-varying information in the Bloom filters, allowing users to be linked across multiple handshakes. Additionally, E-SmallTalker does not apply the Bloom Filter to high-entropy secrets, and thus is vulnerable to an offline dictionary attack.

Sun et al. [51] present a new way of building trust relationships between users by comparing spatiotemporal profiles (log of time and location pairs). In addition to a PSI-based scheme, they present another scheme using Bloom filters that trades off estimation accuracy and privacy in a user-defined manner. In SDDR, we avoid the privacy vs. accuracy trade off since the linkable users share a high-entropy secret as opposed to low-entropy time, location pairs.

Dong et al. [30] use *garbled* Bloom filters to create a practical PSI protocol that handles billions of set members. While more efficient than existing PSI protocols, it does not *scale down* when applied to small set sizes on resource-constrained devices. Because of its reliance on secret shares instead of bits in the Bloom filter, the smallest possible Bloom filter to handle a maximum of 256 items would be 17736 bytes — two orders of magnitude larger than what SDDR requires. In addition, the

communication cost of this interactive protocol increases linearly with the number of nearby devices.

Nagy et al. [45] use Bloom filters to provide a PSI protocol that allows users of online social networks (OSNs) to determine common friends while preserving user privacy. While their solution provides ample efficiency gains over standard PSI, saving an order of magnitude in communication and computation costs, several seconds per *interactive* exchange is too much when running on power-constrained devices in dense environments.

Authenticated key exchange Secure device discovery and recognition should not be confused with mutual authentication, or authenticated key exchange (AKE) protocols [21, 40]. SDDR aims to achieve device discovery and recognition; guaranteeing mutual authentication is not a goal of the basic SDDR protocol. As noted in Section 4.4, once Alice's device recognizes Bob's device, Alice can authenticate Bob by soliciting an explicit verification message from Bob; however, authentication will only be performed if desired by the higher-level application (or user). While secure device discovery and recognition can be achieved by executing an AKE protocol with each nearby device (for all possible shared secrets), such a scheme would be prohibitive in an environment with many nearby peers.

3 Problem Overview

In this section, we review the requirements for a secure encounter protocol, sketch a strawman design, and make observations that enable a practical protocol.

Devices executing a *secure encounter protocol* should detect nearby participating devices, and learn their current ephemeral network identifier. Additionally, each pair of nearby devices should generate a unique (except with negligible probability) shared secret key, known only to the pair. This key allows the devices to: 1) uniquely refer to a particular encounter; and, 2) authenticate each other as the peer in the encounter and securely communicate. The pair should learn no other information about each other; when the same pair of devices meet again, the shared secret and network identifiers exchanged should be unrelated.

By default, devices should remain unlinkable, meaning that no identifying information is exchanged. While unlinkability is appropriate between strangers' devices, friends may wish to enable their devices to recognize each other. A user who allows her device to be recognized by a friend during future discoveries is termed *selectively linkable* (or simply *linkable*) by that friend. When two devices discover each other, a *recognition protocol* should determine if the remote device corresponds to a linkable user. Selectively linkable users must share a unique secret value such that the devices can authenticate each other during the recognition protocol; we refer to this shared secret as the *link value*. Users can derive the link value from the shared secret established during a prior encounter, or using an out-of-band protocol.

In general, users may not wish to be recognizable by their entire set of friends at all times (e.g., Alice may only want her work colleagues to recognize her device while at work). Therefore, a user should be able to contextually (e.g., in terms of time, place, activity) filter the set of friends that can recognize them. This filtering requires that revocation of selective linkability be efficient (e.g., not require a group re-keying) and unilateral (e.g., not require communication). Additionally, the filtering may take place in one direction: Alice may want to not be recognizable by Bob, yet still want to recognize him. Therefore, we consider two distinct sets of link value: the set of advertiseIDs (i.e., who you are willing to be recognized by), and the set of listenIDs (i.e., who you want to recognize). Alice's device is able to recognize Bob's device if and only if their shared link value is in Bob's advertiseIDs and in Alice's listenIDs.

3.1 Security Requirements

We summarize the security requirements below:

Secure communication If Alice and Bob share an encounter, they are able to securely communicate using an untrusted communication channel, both during and after the encounter, and regardless of whether Alice and Bob have opted to selectively link their devices.

Unlinkability The information exchanged during a secure encounter reveals no identifying information about the participating devices, unless the devices have been explicitly linked. In particular, unlinked devices that encounter each other repeatedly are unable to associate their encounters with a previous encounter.

Selective linkability Alice and Bob can optionally agree to be linkable, and therefore able to recognize and authenticate each others' devices in subsequent discoveries.

Revocability Alice may, at any time, *unilaterally* revoke Bob's ability to recognize her.

3.2 Threat Model

We assume that user devices, including the operating system and any applications the user chooses to run, do not divulge information identifying or linking the device or user. Preventing such leaks is an orthogonal concern outside SDDR's threat model. User devices attempt to participate in the protocol with all nearby discovered devices, a subset of which could be controlled by attackers, who may all collude.

We do not consider radio fingerprinting attacks, which detect a device by its unique RF signature [22]. Such

attacks may require sophisticated radio hardware, and are outside our threat model.

3.3 Strawman Protocol

A strawman scheme using existing cryptographic tools, namely Diffie-Hellman [29] (DH) and Private Set Intersection [28, 39] (PSI), can meet the requisite security requirements outlined above. Upon detecting a device, the protocol performs a DH exchange to agree upon a shared secret key. By generating a new DH public and private key pair prior to each exchange, devices remain *unlinkable* across encounters.

To recognize selectively linkable devices, the protocol executes PSI over the devices' advertised and listen identifier sets. *Selective linkability* and *revocability* properties are satisfied by all PSI protocols; however, in order to preserve privacy, we require a PSI protocol that supports *unlinkability* across multiple executions.

While the DH+PSI strawman achieves the desired security properties, it is not practical when frequently run on resource constrained devices. As shown in Section 6, the computation and communication requirements of existing PSI constructions are prohibitively high.

3.4 Observations

In order to enable a practical protocol we rely on several observations:

First, *strict unlinkability* requires that two different discoveries between a pair of devices are unlinkable, regardless of how closely the discoveries are spaced in time. This property cannot be achieved with a non-interactive protocol, because it requires a change of ephemeral network ID and DH keys after each discovery. In order to use a non-interactive protocol, we must settle for the slightly weaker property of *long-term unlinkability*; devices may be linkable within a time epoch, but they remain unlinkable across epochs. For an epoch on the order of minutes, long-term unlinkability is sufficient in practice. It is important to note that epoch boundaries and durations do not require time synchronization; devices may choose when to change epochs independently.

Second, detecting selectively linked devices requires an intersection of the sets of advertised link values between a pair of devices. Even a simple, *insecure* intersection protocol would require the transmission of the complete sets during each pair-wise device discovery, which is too expensive. However, we note that in a large deployment, discoveries among strangers are far more common than discoveries among linked devices. Therefore, an over-approximation of the set intersection may suffice. False positives can be resolved when two presumed linkable devices attempt to authenticate each other using the shared link value.

Finally, we can take advantage of the fact that link values shared between users are high-entropy values taken from a large space, by design. General purpose PSI protocols, on the other hand, ensure security even when sets contain low-entropy values (e.g., dictionary words).

Using these observations, we present the SDDR protocol, which meets the security requirements with practical performance and energy efficiency.

4 SDDR Design

4.1 High-Level Protocol

Like the strawman protocol, SDDR uses DH to exchange a shared secret key with each nearby device; however, SDDR performs the exchange in a non-interactive manner. Periodically, each device broadcasts its DH public key and receives broadcasts from other nearby devices, computing all pair-wise shared secret keys.

SDDR divides time into epochs, during which the ephemeral network address, DH public/private key pair, and advertiseIDs set digest remain constant. Devices are unlinkable *across* epochs, thus preserving *long-term unlinkability*. To avoid expensive synchronous communication, epochs are not synchronized among devices. As a result, the DH computation may fail to produce a shared key if it occurs around an epoch change of either device in a pair. For instance, Alice receives Bob's broadcast in her epoch n, but Bob fails to receive Alice's broadcast until her epoch $n+1$, so he computes a different key. Because broadcasts occur more frequently than epoch changes (seconds versus minutes), however, the probability that a broadcast round yields a *shared* key quickly tends to one with every broadcast round.

Since the link identifiers shared between users are high-entropy values chosen from a large space (e.g., a shared key produced during a prior discovery), SDDR can recognize linkable devices by broadcasting salted hashes of their respective set of advertiseIDs. The DH public key is used as the salt; since the salt is different in each epoch, a device cannot be recognized by the bit-pattern in its Bloom filter across epochs, that ensuring long-term unlinkability. Each user then searches over the hashes using their own set of listenIDs, along with the corresponding salt value, in order to identify the listenID (if any) associated with the remote device.

However, the communication required for moderately-size sets (e.g., 256 advertiseIDs) is still too large for an efficient implementation in Bluetooth due to (pseudo-) broadcast message length constraints. By allowing the recognition protocol to *over-approximate* the actual intersection between the set of local listenIDs and remote advertiseIDs, SDDR can use a probabilistic set digest data structure to reduce the communication needed to determine the intersection. The size of the set digests can be parameterized based on the message size restrictions of

the radio standard used for communication. The choice affects performance only; false positives due to the use of set digests can be resolved using the shared link values, and therefore have no bearing on the protocol's security.

The *selective linkability* property is satisfied by the use of non-deterministic hashes of the link identifiers shared by two users, only allowing linkable users to recognize each other. The *revocation* is supported by the user's ability to add or remove link values from the set of advertiseIDs.

4.2 Formal Problem Definition

We divide the non-interactive SDDR protocol into two algorithms (GenBeacon and Recognize), which we formalize below:

$\mathsf{beacon} \leftarrow \mathsf{GenBeacon}(\mathsf{advertiseIDs})$

In each epoch, a device wishing to participate in peer encounters executes the GenBeacon algorithm, which takes as input the current set of advertiseIDs. The GenBeacon protocol outputs a message beacon, which the device then broadcasts to nearby devices.

$(sk, \mathsf{listenIDs}_{\mathrm{re}}, \mathsf{L}) \leftarrow \mathsf{Recognize}(\mathsf{beacon}_{\mathrm{re}}, \mathsf{listenIDs})$

Upon receiving a $\mathsf{beacon}_{\mathrm{re}}$ from a remote peer, a device executes the Recognize algorithm, which additionally takes in the current set of listenIDs. The Recognize algorithm outputs a secret key sk, the set of $\mathsf{listenIDs}_{\mathrm{re}}$ associated with the remote peer, and the link identifier L for this encounter.

4.3 Detailed Protocol Description

Next, we provide a detailed description of the SDDR protocol. Pseudo-code for the *GenBeacon* and *Recognize* algorithms is shown in Figure 1. In the protocol, as well as our implementation, we use Bloom filters as the probabilistic set digest data structure; however, other set digests (e.g. Matrix filters [46]) could be used instead.

Each user P_i starts by running GenBeacon in order to generate the beacon message to broadcast during the current epoch. GenBeacon first selects a random DH private key α_i, which corresponds to the DH public key g^{α_i}. Afterwards, GenBeacon computes the Bloom filter by hashing each advertiseID within AS_i (the set of advertiseIDs), using the DH public key as the salt. The resulting beacon contains the public key and the Bloom filter.

Each user P_i broadcasts their respective beacon during the epoch. After receiving a beacon from a remote user P_j, user P_i runs the Recognize algorithm. Recognize first computes the DH secret key dhk_{ij}, using the local user's DH private key and the remote user's DH public key (as contained in the beacon). Using the dhk_{ij} along with the local and remote DH public keys, Recognize computes the shared link identifier L_{ij}, which can optionally be used in case the two users wish to selectively link.

Additionally, Recognize computes the key sk_{ij} using the link identifier L_{ij}, which the two devices can use to authenticate each other as the peer associated with this encounter, and then securely communicate. Finally, Recognize queries the Bloom filter by hashing each listenID within LS_i (the set of listenIDs), using the remote user's DH public key as the salt, resulting in the set of matches M_j.

Recall that sk_{ij} may not be shared (i.e., $sk_{ij} \neq sk_{ji}$) in some cases when individual devices decide to change epochs. When a device attempts to communicate using such a key, the authentication will fail, and the device retries with a key produced in a subsequent discovery. Also, to make sure a valid link identifier is used, devices attempt to authenticate each other as part of the pairing process to selectively link.

Notation: Let $\mathsf{BF}\{S\}$ denote a Bloom filter encoding the set S. Let H_0, H_1, and H_2 denote independent hash functions later modeled as random oracles in the proof.

Inputs: Each user P_i has a set of listenIDs (LS_i) and a set of advertiseIDs (AS_i).

Outputs: For all users P_j, discovered by P_i, P_i outputs:

1. sk_{ij}: A shared secret key
2. $\mathsf{listenIDs}_{\mathrm{re}}$: Set of matching $\mathsf{listenID} \in LS_i$ associated with P_j
3. L_{ij}: A shared link identifier

Protocol: Each P_i performs the following steps:

$\underline{\mathsf{GenBeacon}(AS_i)}$

1. Select random $\alpha_i \in_R \mathbb{Z}_p$
2. Compute $\mathsf{BF}_i := \mathsf{BF}\{H_1(g^{\alpha_i}||x) : x \in AS_i\}$
3. Create $\mathsf{beacon}_i = (g^{\alpha_i}, \mathsf{BF}_i)$

Each user P_i broadcasts beacon_i***. For each user P_j that user P_i discovers, P_i runs*** Recognize.

$\underline{\mathsf{Recognize}(\mathsf{beacon}_j, LS_i)}$

1. Compute DH key $\mathsf{dhk}_{ij} = (g^{\alpha_j})^{\alpha_i}$
2. Compute link $\mathsf{L}_{ij} := \begin{cases} H_0(g^{\alpha_i}||g^{\alpha_j}||\mathsf{dhk}_{ij}) \text{ if } g^{\alpha_i} < g^{\alpha_j} \\ H_0(g^{\alpha_j}||g^{\alpha_i}||\mathsf{dhk}_{ij}) \text{ otherwise} \end{cases}$
3. Compute key $sk_{ij} := H_2(\mathsf{L}_{ij})$
4. Query for set $\mathsf{M}_j := \{x : x \in LS_i \wedge H_1(g^{\alpha_j}||x) \in \mathsf{BF}_j\}$

Figure 1: SDDR Non-Interactive Protocol.

Hiding Bloom filter load After receiving multiple Bloom filters, and calculating the distribution of the number of bits set, it is possible to determine the size of the remote user's set of advertiseIDs. This leaks information, which could be used to link devices across multiple epochs. To prevent this leak, the Bloom filters are padded to a global, uniform target number of elements N. Rather than computing actual hashes, we randomly

select $K*(N-|\mathsf{advertiseIDs}|)$ (not necessarily distinct) bits to set to 1, where K is the number of hash functions used in creating the Bloom filter.

4.4 Identification and Authentication

Identification and mutual authentication are not required by all applications, and hence are not a part of the basic SDDR protocol. However, identification and authentication can be achieved easily on top of SDDR as follows:

Identification Identification allows a user to associate a principal (e.g., "Bob") to a specific encounter through the use of out-of-band (OOB) mechanisms. As part of the identification procedure, the users agree on the link identifier (corresponding to the shared encounter) for the purpose of selective linkability. If Alice wishes to be recognizable by Bob in the future, she will insert the link identifier into her advertiseIDs; likewise, if she wishes to recognize Bob in the future, she will insert the link identifier into her listenIDs. However, choosing to enable (or revoke) recognizability is not part of the identification procedure, and can be performed any time by the user after the initial, one-time identification has taken place.

It is well known that achieving secure identification, resistant to man-in-the-middle (MITM) attacks, requires either an *a priori* shared secret or an OOB channel. Any manual authentication technique [32,41,42,53] (e.g., displaying and comparing pictures on both devices, generated from the link identifiers) allows Alice to securely identify Bob's device free of MITM attacks. Additionally, a technique not relying on OOB mechanisms has been proposed by Gollakota et al. [34] for 802.11, using tamper-evident messaging to detect and avoid MITM attacks. Note that many applications do not require identification, such as when users wish to (anonymously) share photos with other event participants.

Mutual authentication Mutual authentication bootstraps a secure and authenticated session between two peers using an *a priori* shared secret (e.g., the link identifier agreed upon as part of the identification procedure). Suppose that in a previous encounter, Alice and Bob participated in the identification procedure; additionally, both Alice and Bob elected to add the shared link identifier to both their advertiseIDs and listenIDs. Thus, in future encounters, Alice and Bob can now authenticate each other (free of MITM attacks). While the basic SDDR protocol does not provide authentication, it can easily be achieved by sending an explicit verification message. For example, a user can prove to a remote peer that they know the common link identifier L by simply sending the verification message $\langle \text{nonce}, H_3(L||\text{nonce}||\mathsf{dhk})\rangle$.

Alternatively, a user can execute a standard authenticated key exchange (AKE) protocol; however, in the case of SDDR, since a DH key is already exchanged, an explicit verification message is sufficient and cheaper than a standard AKE protocol. Mutual authentication only needs to be performed when requested by an application (or user), and thus is not part of the base SDDR protocol.

4.5 Suppressing Bloom filter false positives

The false positive probability of a Bloom filter, denoted as P_{fp}, is computed as a function of: the number of elements inserted (N), the size (in bits) of the Bloom filter (M), and the number of hash functions per element (K). Although Bose et al. [47] provide a more accurate (yet not closed form) solution, P_{fp} is closely approximated by the following formula:

$$P_{\mathsf{fp}} = \left(1 - \left[1 - \frac{1}{M}\right]^{KN}\right)^{K}$$

In the SDDR protocol, these false positives manifest themselves as selectively linkable principals associated with the remote device (and their current shared encounter). By default, false positives are not reduced over the course of an epoch, and only mutual authentication (see Section 4.4) will allow two peers to check if they are selectively linkable (resolving any false positives). Ideally, we want to provide a way for the matching set (M_j in the protocol) to converge towards the exact intersection of the remote peer's advertiseIDs and the user's listenIDs over time.

If one creates multiple Bloom filters, each using a unique set of hash functions (or salt value(s)), the intersection of the matching sets for the Bloom filters results in an overall matching set with a reduced false positive rate. Within a single epoch, a device can compute and distribute beacons with unique Bloom filters that evolve over time. Since beacons within the same epoch use the same DH public key, we modify Step 2 within the GenBeacon algorithm to additionally use an incrementing counter *count* as part of the salt:

$$\mathsf{BF}_i := \mathsf{BF}\{H_1(g^{\alpha_i}||x||\mathit{count}) : x \in AS_i\}$$

count increments each time a beacon is broadcast, and is reset to 0 at the start of every epoch. We use this extension as part of our implementation, as described in Section 5.

5 SDDR on Android

We have implemented the SDDR protocol on the Android platform as part of a system service. The codebase is written in C++ and runs with root privileges[1] . For development and evaluation, we use Samsung Galaxy

[1] SDDR requires root privileges to handle address changes, as well as to support efficient communication through Extended Inquiry Response (EIR) payloads.

Nexus phones running Android 4.1.2 [2] with the android-omap-tuna-3.0-jb-mr0 kernel. For our implementation, we selected to use Bluetooth for short-range radio communication; other short-range radios (e.g. WiFi, ZigBee) could also be used for this purpose. We selected Bluetooth 2.1 (BT2.1) over BT4 because a BT2.1 implementation closely mirrors the protocol as described; however, we designed a BT4 implementation for use in EnCore [17]. We use elliptic curve cryptography (ECC) due to the smaller key sizes relative to RSA, selecting the 192-bit curve as recommended by NIST [4].

We first describe the implementation of the major components of the SDDR protocol: discovery, handshake, and epoch change. Afterwards, we briefly describe the system service that we developed to allow all applications running on the device to take advantage of SDDR, without each independently managing discoveries. Finally, we discuss our integration of the system service with the open-source Haggle framework [8].

5.1 SDDR Protocol Components

Discovery In the protocol, a single beacon is broadcast throughout each epoch. In our implementation, since devices must wake up to discover nearby devices and receive their beacon messages, we break down each epoch into multiple discovery intervals. Using the protocol extension described in Section 4.5 to reduce Bloom filter false positives, we generate and broadcast a new beacon during every discovery interval.

There are two roles that devices can take on within the Bluetooth 2.1 discovery protocol: *discoverable* and *inquirer*. Every device always plays the role of *discoverable*, responding to incoming inquiry scans with information on how the *inquirer* can establish a connection (e.g., MAC address). By using the extended inquiry response (EIR) feature present in BT2.1, which includes an additional 240 byte payload added to the response, *discoverable* devices can transmit their beacon to the discoverer during the inquiry scan itself.

At the start of each discovery interval, a device additionally takes on the role of *inquirer*, performing an inquiry scan in order to collect and process beacon messages from nearby devices. In addition, the device will update its EIR payload with a new beacon message; this payload will be used as a response while *discoverable*.

Devices must only wake up when acting as a *inquirer*. Otherwise, while simply *discoverable*, only the Bluetooth controller (and not the main CPU) must be active; the controller wakes up every 1.28 seconds to listen and respond to inquiry scans from nearby devices.

[2]The Android 4.4 release would provide additional energy savings with respect to suspend and wakeup transitions due to the updated AlarmManager API.

Compute keys and recognize When a *inquirer* detects a new device, which could also be an epoch change by an existing device, it computes the shared secret for the current epoch using the local DH private key and the remote device's DH public key (embedded in the beacon). For each device: 1) for its first beacon, the *inquirer* queries the Bloom filter contained in the beacon using $H_1(g^{\alpha_i}||x||count)$ for each x in its set of listenIDs; 2) for subsequent beacons, the *inquirer* queries the Bloom filter only for each x previously determined to be in the intersection.

Periodic MAC address change SDDR ensures that the discovery and recognition protocol does not leak linkable information. However, the underlying Bluetooth packets have a static MAC address that can be used to track the device (and the user). As part of our Bluetooth implementation, we choose a random Bluetooth MAC address at the start of every epoch. BT2.1 does not provide a native interface for changing MAC addresses "on-the-fly"; therefore, we reset the Bluetooth controller each time the address is changed (once per epoch, nominally fifteen minutes). Unfortunately, this reset closes ongoing connections and invalidates existing device pairings; however, the BT4 specification supports changing the public (random) address for the device while maintaining the private address for paired devices.

5.2 SDDR Integration

We chose to implement the SDDR protocol as part of a system service on Android. The centralized service allows for greater energy efficiency as it can broadcasts discovery information to all applications via IPC mechanisms, as opposed to each application performing its own discovery. Currently, we allow applications to connect to the service via local Unix sockets. Applications receive messages for each discovered device, along with the shared secret and identity information (if selectively linkable) for SDDR-aware devices.

Haggle Haggle is a mobile communication platform for device-to-device radio communication, and supports a number of content sharing apps. A photo sharing app, for instance, shares with nearby devices photos whose textual attributes match a user's specified interests.

To demonstrate some of SDDR's capabilities, we have modified Haggle to use SDDR. This enables Haggle and its applications to communicate securely with nearby devices, without revealing any linkable information and without the risk of tracking by third parties. We have modified Haggle's photo sharing app to take advantage of SDDR's features. Users can add a special attribute to a photo, which narrows its visibility to a specified set of linkable user(s). If a photo carries this attribute, it is eligible for sharing only with devices of linkable users in

this set. Finally, when a photo is shared, it is encrypted with the shared key established by SDDR.

6 Experimental Results

In order to evaluate the design and implementation of SDDR, we first perform a comparison to the strawman protocol by focusing on the PSI portion in comparison to our Bloom filter-based approach. Secondly, we look at energy consumption both at the level of benchmarking individual operations within the SDDR (and other) protocols, as well as battery life consumption over the course of a day. Additionally, we attempt to analyze the scalability (and DoS resistance) of the SDDR protocol by extrapolating energy consumption results to a large number of devices. For a more application-level evaluation, we refer the reader to our work on EnCore [17], which includes a deployment with 35 users, using a BT4 implementation of SDDR.

6.1 Comparison with PSI

We measure the SDDR discovery protocol computation time while varying the number of linkable identifiers, and compare the elapsed time to that required for a PSI protocol. We use an implementation [11] of the JL10 scheme [38], one of the fastest schemes known-to-date. JL10 is secure and can be modified to achieve unlinkability across sessions. Both protocols are executed using a single core on the 1.2 GHz ARM Cortex-A9 processor.

Figure 2 shows the run times for each protocol; each bar is an average of 50 runs, with error bars denoting the 5^{th} and 95^{th} percentile values. We divide SDDR into two separate trials, varying the number of advertisements in order to achieve the specified false positive rates for each trial. Additional advertisements do not require much computation time because SDDR only uses the complete set of listen IDs for the first Bloom filter; afterwards, SDDR uses the matching set of listen IDs, which quickly converges towards the actual intersection.

Results show that SDDR is up to *four orders of magnitude* faster than standard PSI. The gain in performance is crucial for practical deployment, as these computations take place for every discovered device. In order to preserve user privacy against tracking, large maximum set sizes (128 to 256 entries) with random entry padding must be used with typical PSI protocols; alternatively, a size-hiding PSI scheme [18] can be used, but the performance in practice is worse than the scheme we used [10].

6.2 Energy Consumption

SDDR runs on resource-constrained devices, therefore we evaluate its energy consumption in detail. First, we look at microbenchmarks for the individual components of the protocol (e.g., discovery), as well as various idle states, which provide a baseline for energy consumption.

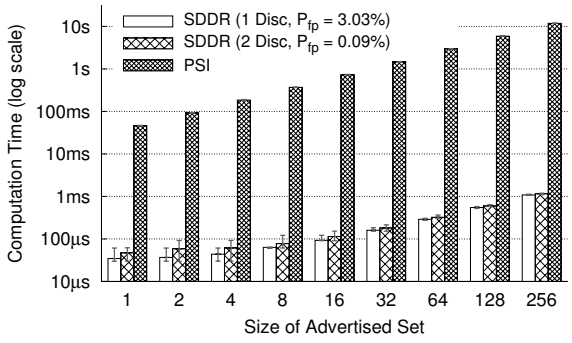


Figure 2: Protocol execution times of PSI versus SDDR for an encounter with varying sizes of advertised sets of link values. The "# disc" represents the number of discovery beacons used to compute the matching set, with P_{fp} as the associated false positive probability.

Second, we collect and analyze power traces of our protocol over several epochs in order to determine the battery life cost of frequently running our handshake protocol over the course of a day. Third, we estimate the reduction in battery consumption when in densely crowded areas, or under denial of service attacks, a device discovers the specification maximum of 255 devices per inquiry scan [3]. In order to monitor energy consumption over time, we use the BattOr [49] power monitor.

Microbenchmarks In Table 2, we outline the results from the microbenchmark experiments. We collect 25 data points for each experiment, and present the average values in the table above. We enable airplane mode on the device for each test, ensuring that all radio interfaces are disabled unless otherwise explicitly requested. Idle state requires very little power, as the device remains in suspended state with the main processor powered off. Since epoch changes require disabling and re-enabling the hardware Bluetooth controller, the controller requires several seconds to return to its working state. An epoch change requires 568mJ energy consumption — however, note that epoch changes are relatively infrequent (e.g., every fifteen minutes) compared to discovery.

Additionally, we collected power traces for various discovery and recognition protocols. When there are no nearby devices, the baseline discovery protocol in Bluetooth 2.1 costs 1363mJ per discovery. In comparison, our implementation of the SDDR protocol over Bluetooth 2.1 incurs only 7% additional energy cost while executing the recognition protocol with 5 nearby devices (and using 256 advertised and listen IDs). The implementation of the DH+PSI strawman over Bluetooth 2.1 requires much more energy per execution, over an order of magnitude greater than the baseline (43,335mJ compared to 1,363mJ). This is expected as the PSI protocol

Component	Avg. Power (mW)	Energy (mJ)
Idle	1.73	-
Bluetooth 2.1		
Discovery (0 Devices)	118	1,363
Incoming Connection	200	893
Discovery and Recognition (5 Devices, 256 Listen IDs)		
SDDR over BT2.1	124	1,464
DH+PSI over BT2.1	404	43,335
ResolveAddr in BT4.0	226	737
SDDR Epoch Change	178	568

Table 2: Average power and energy consumed by various components, or system states. Components which have energy consumption marked as '-' have no well defined duration.

State	Energy (J)	Battery (%)
Full Battery	23,328	100
Idle	150	0.64
Idle with Bluetooth	188	0.81
Running (5 Devices, 256 Listen IDs) (60s Discovery Interval)		
SDDR over BT2.1	2,511	10.76
ResolveAddr in BT4.0	1,260	5.40
DH+PSI over BT2.1	44,619	191.27
IncConn over BT2.1	9,143	39.19
(120s Discovery Interval)		
SDDR over BT2.1	1,300	5.57
ResolveAddr in BT4.0	718	3.08
DH+PSI over BT2.1	35,097	150.45

Table 3: Energy and battery life consumption for different states and protocol configurations over the course of one full day. A daily battery consumption of $p\%$ means that the battery would last $100/p$ days if the device runs the corresponding protocol and is otherwise idle.

is not as efficient as SDDR in terms of computation (See Figure 2) and communication, and it must execute an interactive protocol for each nearby device.

The ResolveAddr protocol, implemented as per the Bluetooth 4.0 specification, requires less energy (737mJ) compared to other schemes; however, it neither exchanges a session key, nor supports efficient revocation of the set of linked users [3]. ResolveAddr is optimized to support a limited feature set, and uses the efficient broadcast channels made available in Bluetooth 4.0.

In addition, as a point of comparison between interactive and non-interactive protocols, we collected power traces for a device waking up to handle an incoming connection over Bluetooth 2.1 (without performing any work). This incoming connection consumes an average of 893mJ, which is roughly 65% of the cost of an entire discovery operation. This connection cost scales linearly, which makes interactive protocols impractical for handling many nearby devices.

Reduction in battery life In order to gauge the reduction in battery life of frequently running a discovery and recognition protocol, we collected power traces for various protocol configurations with up to 5 nearby devices over the course of two epochs (30 minutes). For each protocol, we evaluate two different discovery intervals (60 and 120 seconds); existing applications, such as Haggle [8], use a 120 second interval. Since the energy consumption remains the same across two epochs, we extrapolate the energy consumed to a full day (24 hour period), as shown in Table 3.

The Samsung Galaxy Nexus battery has a capacity of 6.48Wh, which we convert to $23,328$J for the purpose of comparisons within the table. With 5 nearby devices, SDDR uses 5.57% of the battery life per day with a 120 second discovery interval; ResolveAddr uses slightly less than SDDR (around 3%), due to the reduced discovery costs. In comparison, the DH+PSI protocol consumes around 150% of the battery over the course of 24 hours. This means that the battery would completely drain within 16 hours, or within only 12.6 hours when using the 60 second discovery interval. IncConn provides a point of reference for the base-line battery life of an interactive protocol—without executing any protocol, it consumes around four times as much energy as SDDR.

As previously mentioned, we assume that each discovery returns 5 *new* nearby devices; in the case of SDDR, this requires computing the shared secret and using the complete set of listen IDs (instead of the matching set) to query the received Bloom filter. In practice, there will not always be 100% churn in nearby devices in each discovery period, meaning that these results are *conservative estimates* of actual energy consumption.

In order to provide a visual comparison between the protocols, we present snapshots of a single 120 second discovery interval for both the SDDR and DH+PSI protocols in Figure 3. These snapshots show the power consumed at each point in time; energy consumption is computed by integrating over a given interval of time. Both protocols initiate a discovery at around the 20 second mark. Since we designed the SDDR protocol to support non-interactive execution, SDDR over BT2.1 can take advantage of executing both the discovery and recognition portions at the same time. Unlike SDDR, the DH+PSI protocol must perform an interactive recognition protocol that takes longer than the discovery process itself, and must be performed individually with each nearby device. In the right half of the plot, both devices handle incoming discovery and recognition requests from nearby devices (5 for SDDR, and 1 for

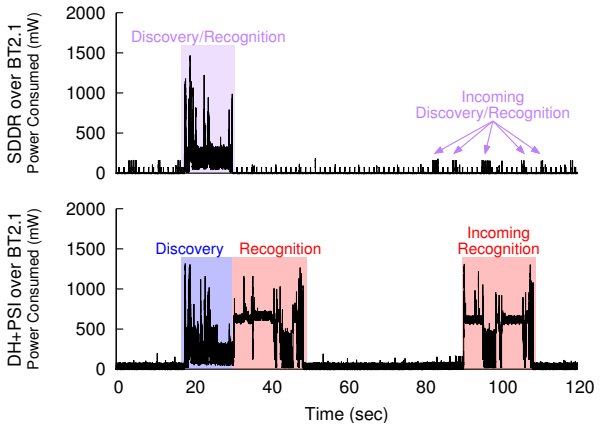


Figure 3: Power traces from running the SDDR and DH+PSI protocols (implemented over Bluetooth 2.1) for one discovery interval of 120 seconds.

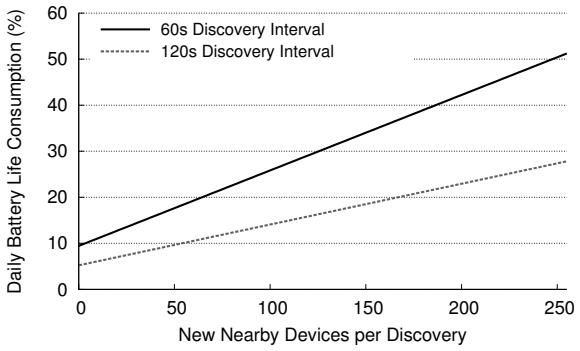


Figure 4: Estimated daily battery life consumption while running the SDDR protocol, for varying numbers of nearby devices.

DH+PSI). Even for the case of a single nearby device, DH+PSI is not practical.

Crowds and DoS attacks A frequently running protocol such as SDDR can potentially open up a new avenue of attack, whereby attackers can try to exhaust the battery of a victim device by forcing it to continually perform new discoveries. Even in benign scenarios, a device may legitimately perform many discoveries over a prolonged interval, e.g., when the user is at a stadium or an indoor auditorium, and the device encounters many other Bluetooth enabled devices. In this section, we experiment with these extreme scenarios, and show that SDDR does not adversely affect battery consumption, regardless of the number of peers it discovers. At the same time, SDDR is able to discover linked peers, and provide reasonable performance even in crowded spaces.

In order to study these worst-case scenarios for SDDR, we estimate the reduction in battery life assuming that there are 255 *new* device responses for every inquiry scan we perform. We use three components of energy consumption for our estimate: idle with Bluetooth running in discoverable mode (E_{BT}), epoch changes (E_{EC}), and discoveries (E_D). Each of these components represents the amount of energy consumed in mJ over the course of a full day (24 hours), and together represent the aggregate energy consumption while running the SDDR protocol:

$$E_{SDDR}(d,i) = E_{BT}(d,i) + E_{EC} + E_D(d,i) \quad (1)$$

$E_{BT}(d,i)$ and $E_D(d,i)$ vary with respect to the number of nearby devices (d), and the discovery interval in seconds (i). We measured the average energy consumption for cases of 1, 3, and 5 discovering devices, for discovery intervals of 60 and 120 seconds. Additionally, we use the results of the microbenchmarks from Table 2 to provide formulas for the energy consumed by epoch changes and discoveries. We assume a linear model, with respect to the number of nearby devices (d), for each i (either 60 or 120 seconds) in computing the formulas for $E_{BT}(d,i)$ and $E_D(d,i)$. The energy consumed by epoch changes (E_{EC}) does not vary with respect to d or i.

We validate Equation 1 by comparing its results to the measured values from Table 3. Without any nearby devices, our estimates are off by 1.96% and 0.66% for 60 and 120 second discovery intervals respectively. Likewise for the case of 5 devices, our estimates deviate from the measurement results by 4.61% and 1.74%.

The estimated daily battery life consumption for varying numbers of nearby devices is shown in Figure 4. Over the course of an entire day, running the SDDR protocol with a 120 second discovery interval consumes 27.82% battery life in this worst-case scenario.

Comparison with SmokeScreen We compare the performance of our protocol with SmokeScreen's discovery protocol [27]. SmokeScreen requires sending one clique signal per advertised ID, and does not use a set-digest data structure (e.g. Bloom filter) to aggregate them. In the authors' implementation, the clique signals are sent over a Bluetooth name request, which holds 248 bytes of data, i.e., roughly 4 clique signals. This makes SmokeScreen less scalable with larger advertised sets: 1) for more than 4 advertised IDs, the clique signals have to be sent over multiple Bluetooth name requests (increasing the discovery latency), and 2) sending multiple name requests for large advertised sets also lead to additional energy consumption. SDDR requires a constant amount of time to detect linkability to a given false positive rate, while SmokeScreen's detection time increases linearly with respect to the number of clique signals.

Since the SmokeScreen measurements were reported several years back, we re-evaluated the energy consumption for SmokeScreen on a recent device. In our measurement, we *conservatively* estimate the energy consump-

tion for SmokeScreen, by measuring only communication related, but not computation- or storage-related energy overhead. Our results suggest that in the case of one nearby device, SmokeScreen's communication consumes $1,628$mJ of energy; for three devices, this increases to $2,071$mJ. Unfortunately, the cost of performing a name request for each individual nearby device is prohibitively expensive. For the same amount of energy spent by SmokeScreen with 3 nearby devices, SDDR can discover and process 35 nearby devices (from Equation 1).

7 Discussion

In this section, we discuss properties and implications of the somewhat unique communication model provided by SDDR, which departs from the norm in two basic ways:

- SDDR decouples confidentiality from identity: encounter peers can communicate securely, even though they do not know each other, and cannot recognize each other during future encounters.
- Communication within SDDR is both defined and limited by radio range, which may not necessarily conform to application semantics.

7.1 Confidentiality without Identity

SDDR's secure encounter primitive provides, in effect, a per-encounter mutual pseudonym for the encounter peers, and an associated shared key. It enables the peers to name each other and communicate securely during their encounter, and at any time after their encounter via an untrusted rendez-vous service. The peers can name and authenticate each other as participants in a specific encounter and communicate securely, while remaining anonymous and unlinkable otherwise (assuming they do not reveal linkable information within their communication). Interestingly, if the users choose, this type of anonymous interaction during an encounter can form the basis for mutual identification and authentication.

Prior systems rely on anonymous or unlinkable encounters between peers, such as SMILE [43] which supports finding missed connections, and SmokeScreen [27] which allows two anonymous peers to exchange addresses for further communication (e.g., E-mail addresses) through the use of a trusted third party service.

7.2 Radio-Range Limited Communication

SDDR communication is limited to radio range, nominally 10 meters for Bluetooth 2.1 (50 meters for the latest Bluetooth 4.0 standard). From an application design point-of-view, range-limited communication may inhibit, but can also prove useful.

Without a third-party data repository or additional protocol mechanisms, e.g., a multi-hop structure, applications that provide notifications among devices beyond radio range cannot be implemented. For example, SDDR cannot be used to replicate the functionality of Google Latitude, which provides updates on friend locations independent of physical distance.

Yet another problem may be that the radio range is not limiting enough! For instance, consider an application that wants to create pairwise encounters and share a group secret only between users who are in the same room. Nothing within SDDR will prevent messages from being received outside the room, enabling a passive eavesdropper to learn the group secret. External mechanisms, in case of the room limited communication, possibly a fast attenuating ultrasound identification beacon are required to manage the impedance mismatch between application semantics and radio range. In general, application designers may choose to use SDDR for a base level of peer detection, and impose criteria that filters unwanted peers.

However, we note that radio range limited communication can have beneficial effects as well. In many situations, the Bluetooth radio range includes the attendees of a socially meaningful event—those with whom a user is likely to interact or share an experience.

Finally, the confidentiality and anonymity provided by SDDR may disproportionately empower abusive users, who could, e.g., spam or otherwise abuse those who are nearby. Here, radio range limited communication provides both a bound on abusive communication and a rudimentary form of accountability. If SDDR is used for malice, the victim is assured that the source of the communication is nearby. The victim could move, or provide evidence of misbehavior (received messages) to law-enforcement authorities. The physical proximity (either of the sender or an accomplice) required for communication within SDDR can potentially serve as a deterrent to abusive communication.

8 Conclusion

In this paper, we articulate the need for efficient secure mobile encounters and their requirements, including selective linkability and efficient revocation. We propose a light-weight protocol called SDDR, which provably meets the security requirements under the random oracle model, and enables highly scalable and energy-efficient implementations using Bluetooth. Experimental results show that our protocol outperforms standard Private Set Intersection by four orders of magnitude. Additionally, its energy efficiency exceeds that of SmokeScreen by an order of magnitude, while supporting stronger guarantees. Energy consumption (and the resulting battery life)

remain practical even under worst-case conditions like dense crowds or DoS attacks.

Acknowledgments

We thank the anonymous reviewers and Jianqing Zhang for their helpful comments, as well as the Max Planck Society. We also thank Rohit Ramesh for his help in integrating SDDR with Haggle. This work was partially supported by the U.S. National Science Foundation under awards IIS-0964541 and CNS-1314857, a Sloan Research Fellowship, as well as by Google Research Awards.

References

[1] AllJoyn. http://www.alljoyn.org.

[2] Bluetooth 2.0 Specification. https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=40560.

[3] Bluetooth Specification Core Version 4.0. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737.

[4] Federal Information Processing Standards Publication: Digital Signature Standard (DSS). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[5] FireChat. http://www.opengarden.com/firechat.

[6] Foursquare. https://foursquare.com/.

[7] Google Latitude. http://www.google.com/latitude.

[8] Haggle. http://www.haggleproject.org.

[9] LoKast. http://www.lokast.com.

[10] Personal communication with Emiliano De Cristofaro.

[11] Privacy-preserving Sharing of Sensitive Information Toolkit. http://sprout.ics.uci.edu/projects/iarpa-app/code/psst-psi.tar.gz.

[12] StickNFind. https://www.sticknfind.com/.

[13] Tile. http://www.thetileapp.com/.

[14] Venmo. https://venmo.com/.

[15] Whisper. http://whisper.sh/.

[16] Google Fires Engineer for Violating Privacy Policies. http://www.physorg.com/news203744839.html, 2010.

[17] ADITYA, P., ERDÉLYI, V., LENTZ, M., SHI, E., BHATTACHARJEE, B., AND DRUSCHEL, P. EnCore: Private, Context-based Communication for Mobile Social Apps. In *MobiSys* (2014).

[18] ATENIESE, G., DE CRISTOFARO, E., AND TSUDIK, G. (If) Size Matters: Size-hiding Private Set Intersection. In *PKC* (2011).

[19] BAKER, L. B., AND FINKLE, J. Sony PlayStation Suffers Massive Data Breach. http://www.reuters.com/article/2011/04/26/us-sony-stoldendata-idUSTRE73P6WB20110426, 2011.

[20] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13* (1970).

[21] BRESSON, E., CHEVASSUT, O., AND POINTCHEVAL, D. Provably Secure Authenticated Group Diffie-Hellman Key Exchange. *TISSEC 10*, 3 (July 2007).

[22] BRIK, V., BANERJEE, S., GRUTESER, M., AND OH, S. Wireless Device Identification with Radiometric Signatures. In *MobiCom* (2008).

[23] BRODER, A., AND MITZENMACHER, M. Network Applications of Bloom Filters: A Survey. *Internet Mathematics 1* (2004).

[24] CALANDRINO, J. A., KILZER, A., NARAYANAN, A., FELTEN, E. W., AND SHMATIKOV, V. "You Might Also Like: " Privacy Risks of Collaborative Filtering. S&P.

[25] CHAMPION, A. C., ZHANG, B., TENG, J., AND YANG, Z. D-Card: A Distributed Mobile Phone Based System for Relaying Verified Friendships. In *INFOCOM WKSHPS* (2011).

[26] CHANG, Y.-C., AND MITZENMACHER, M. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *Applied Cryptography and Network Security* (2005).

[27] COX, L. P., DALTON, A., AND MARUPADI, V. SmokeScreen: Flexible Privacy Controls for Presence-sharing. In *MobiSys* (2007).

[28] CRISTOFARO, E. D., KIM, J., AND TSUDIK, G. Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model. In *ASIACRYPT* (2010).

[29] DIFFIE, W., AND HELLMAN, M. E. New Directions in Cryptography. *IEEE Transactions on Information Theory* (1976).

[30] DONG, C., CHEN, L., AND WEN, Z. When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol. In *SIGSAC* (2013).

[31] FU, Y., AND WANG, Y. BCE: A Privacy-preserving Common-friend Estimation Method for Distributed Online Social Networks without Cryptography. In *CHINACOM* (2012).

[32] GEHRMANN, C., MITCHELL, C. J., AND NYBERG, K. Manual Authentication for Wireless Devices. *RSA Cryptobytes 7*, 1 (Spring 2004), 29–37.

[33] GOH, E.-J. Secure Indexes. *IACR Cryptology ePrint Archive* (2003).

[34] GOLLAKOTA, S., AHMED, N., ZELDOVICH, N., AND KATABI, D. Secure In-Band Wireless Pairing. In *USENIX Security* (2011).

[35] GREENSTEIN, B., MCCOY, D., PANG, J., KOHNO, T., SESHAN, S., AND WETHERALL, D. Improving Wireless Privacy with an Identifier-free Link Layer Protocol. In *MobiSys* (2008).

[36] GRUTESER, M., AND GRUNWALD, D. Enhancing Location Privacy in Wireless LAN Through Disposable Interface Identifiers: A Quantitative Analysis. In *WMASH* (2003).

[37] HU, Y.-C., AND WANG, H. A Framework for Location Privacy in Wireless Networks. In *SIGCOMM Asia Workshop* (2005).

[38] JARECKI, S., AND LIU, X. Fast Secure Computation of Set Intersection. In *SCN* (2010).

[39] JARECKI, S., AND SAXENA, N. Authenticated Key Agreement with Key Re-use in the Short Authenticated Strings Model. In *SCN* (2010).

[40] KATZ, J., AND YUNG, M. Scalable Protocols for Authenticated Group Key Exchange. *J. Cryptol. 20*, 1 (Jan. 2007).

[41] LAUR, S., AND NYBERG, K. Efficient Mutual Data Authentication Using Manually Authenticated Strings. In *CANS* (2006).

[42] LIN, Y.-H., STUDER, A., HSIAO, H.-C., MCCUNE, J. M., WANG, K.-H., KROHN, M., LIN, P.-L., PERRIG, A., SUN, H.-M., AND YANG, B.-Y. SPATE: Small-group PKI-less Authenticated Trust Establishment. In *MobiSys* (2009).

[43] MANWEILER, J., SCUDELLARI, R., AND COX, L. P. SMILE: Encounter-based Trust for Mobile Social Services. In *CCS* (2009).

[44] MOHAISEN, A., FOO KUNE, D., VASSERMAN, E., KIM, M., AND KIM, Y. Secure Encounter-based Mobile Social Networks: Requirements, Designs, and Tradeoffs. *IEEE TDSC* (2013).

[45] NAGY, M., DE CRISTOFARO, E., DMITRIENKO, A., ASOKAN, N., AND SADEGHI, A.-R. Do I Know You?: Efficient and Privacy-preserving Common Friend-finder Protocols and Applications. In *Proceedings of the 29th Annual Computer Security Applications Conference* (2013).

[46] PORAT, E. An Optimal Bloom Filter Replacement Based on Matrix Solving. *CoRR abs/0804.1845* (2008).

[47] PROSENJIT BOSE AND HUA GUO AND EVANGELOS KRANAKIS AND ANIL MAHESHWARI AND PAT MORIN AND JASON MORRISON AND MICHIEL SMID AND YIHUI TANG. On the false-positive rate of Bloom filters. *Information Processing Letters 108*, 4 (2008), 210 – 213.

[48] RASHID, F. Y. Epsilon Data Breach Highlights Cloud-Computing Security Concerns. http://www.eweek.com/c/a/Security/Epsilon-Data-Breach-Highlights-Cloud-Computing-Security-Concerns-637161/, 2011.

[49] SCHULMAN, A., SCHMID, T., DUTTA, P., AND SPRING, N. Demo: Phone Power Monitoring with BattOr. In *MobiCom* (2011).

[50] SU, J., SCOTT, J., HUI, P., CROWCROFT, J., DE LARA, E., DIOT, C., GOEL, A., LIM, M. H., AND UPTON, E. Haggle: Seamless Networking for Mobile Applications. In *Ubicomp* (2007).

[51] SUN, J., ZHANG, R., AND ZHANG, Y. Privacy-preserving Spatiotemporal Matching. In *INFOCOM* (2013).

[52] THOMAS, K. Microsoft Cloud Data Breach Heralds Things to Come. http://www.pcworld.com/article/214775/microsoft_cloud_data_breach_sign_of_future.html, 2010.

[53] VAUDENAY, S. Secure Communications Over Insecure Channels Based on Short Authenticated Strings. In *CRYPTO* (2005).

[54] WATANABE, C., AND ARAI, Y. Privacy-Preserving Queries for a DAS Model Using Encrypted Bloom Filter. In *Database Systems for Advanced Applications* (2009).

[55] YANG, Z., ZHANG, B., DAI, J., CHAMPION, A. C., XUAN, D., AND LI, D. E-SmallTalker: A Distributed Mobile System for Social Networking in Physical Proximity. In *ICDCS* (2010).

[56] ZHU, F., MUTKA, M., AND NI, L. PrudentExposure: A Private and User-centric Service Discovery Protocol. In *PerCom* (2004).

A Formal Security Definitions and Proofs

A.1 Overview

We follow a standard game-based approach for defining security. We describe a game between an adversary and challenger. The adversary controls the communication medium, and is allowed to schedule the actions of legitimate users. For example, the adversary can instruct a legitimate user to run GenBeacon to generate a discovery beacon; or instruct a recipient to receive the beacon(s) and call Recognize to determine the linkability of discovered neighbors. The adversary can also instruct a legitimate user to perform handshake with any member of the compromised coalition. Link identifiers generated during such a handshake (with the adversary) are marked as compromised (i.e., known to the adversary). In addition, the adversary can explicitly compromise an encounter between two legitimate users in which case the secret link identifier and shared key are exposed; or explicitly compromise a user in which case all its internal states, including previous link identifiers, are exposed.

At some point during the game, the adversary will issue a challenge, either an *anonymity challenge* or a *confidentiality challenge*.

An *anonymity challenge* intuitively captures the notion that an adversary cannot break a legitimate user's anonymity, unless the legitimate user has authorized linkability to a party within the adversary's coalition. Note that this part of the definition captures the unlinkability, selective linkability, and revocability requirements (See Section 3.1) simultaneously.

A *confidentiality challenge* intuitively captures the notion that an eavesdropping cannot learn anything about the (online or post-hoc) communication in between two legitimate users. This is guaranteed since for any two users that remain uncompromised at the end of the security, their shared key established for some time epoch t is as good as "random" to the adversary (assuming their encounter in time epoch t also remains uncompromised by the end of the security game).

A.2 Formal Security Definitions

We define the following security game between and an adversary $\mathscr{A}$ and a challenger $\mathscr{C}$. The time epoch t is initialized to 0 at the beginning of the game. The adversary adaptively makes a sequence of queries as below.

Next time epoch. Increments the current time epoch t.

Expose handshake beacons. The adversary specifies an uncompromised user P_i, identifiers of a subset S_i of P_i's previous encounters, and asks the challenger to expose P_i's handshake beacon in the current time step t using the subset of previous encounters S_i.

Handshake - Uncompromised users. The adversary specifies two uncompromised users P_i and P_j, such that P_j can hear P_i in the current time epoch t. After receiving P_i's handshake beacon, P_j calls the Recognize algorithm, and updates its local state accordingly. The adversary does not obtain information from the challenger for this query.

Handshake - Adversary. The adversary sends a handshake beacon to an uncompromised user P_i. P_i calls the Recognize algorithm, and updates its local state. The identifier of this encounter is marked as compromised. The adversary does not obtain any information from the challenger for this query.

Compromise - Encounter. The adversary specifies a reference to an encounter which took place in time $t' \leq t$ between two uncompromised users P_i and P_j, and the challenger reveals to the adversary the corresponding link identifier, encounter key, and any additional information associated with this encounter.

Compromise - User. The adversary specifies an uncompromised user P_i. The adversary learns all P_i's internal state, including the list of all previous link identifiers, encounter keys, received beacons, and any additional information associated with P_i's previous encounters[3]. P_i and all of its link identifiers are marked as compromised.

Challenge. There can only be one challenge query in the entire game, of one of the following types. In both cases, the adversary outputs a guess b' of b selected by the challenger.

- **Anonymity.** Adversary specifies two users P_i and P_j who must remain uncompromised at the end of the game. The adversary specifies S_i and S_j to the challenger, which (respectively) denote a subset of P_i's and P_j's previous encounters that must remain uncompromised at the end of the game. We require that $|S_i| = |S_j|$. Furthermore, at the end of the game, the adversary must not have issued an "expose handshake beacon" query in the current time step for P_i (or P_j) involving any element in the subset S_i (or S_j).

  The challenger flips a random coin b. If $b = 0$, the challenger constructs P_i's handshake beacon for the current time epoch t for the set S_i, and returns it to adversary. If $b = 1$, the challenger constructs P_j's handshake beacon for the set S_j, and returns it to adversary.
- **Confidentiality.** The adversary specifies two users P_i and P_j who must remain uncompromised at the end of the game.

[3] Specific to our construction, the internal states also include the exponents of P_i's own DH beacons in all previous time epochs.

Furthermore, the encounter between P_i and P_j during time epoch t must also remain uncompromised at the end of the game.

The challenger flips a random coin b. If $b = 0$, challenger returns the encounter key sk_{ij} established between P_i and P_j in time epoch t. If $b = 1$, challenger returns a random number (from an appropriate range).

Definition 1 (Anonymity, Selective linkability)**.** *Suppose that the adversary $\mathscr{A}$ makes a single anonymity challenge in the above security game. The advantage of such an adversary $\mathscr{A}$ is defined as $\mathsf{Adv}^{\mathsf{link}}(\mathscr{A}) := |\Pr[b' = b] - \frac{1}{2}|$. We say that our handshake protocol satisfies selective linkability, if the advantage of any polynomially bounded adversary (making an anonymity challenge) in the above game is a negligible function in the security parameter.*

Definition 2 (Confidentiality)**.** *Suppose that the adversary $\mathscr{A}$ makes a single confidentiality challenge in the above security game. The advantage of such an adversary $\mathscr{A}$ is defined as $\mathsf{Adv}(\mathscr{A})^{\mathsf{conf}} := |\Pr[b' = b] - \frac{1}{2}|$. We say that our handshake protocol satisfies confidentiality, if the advantage of any polynomially bounded adversary (making a confidentiality challenge) in the above game is a negligible function in the security parameter.*

A.3 Proofs of Security

Theorem 1 (Anonymity, selective linkability)**.** *Assume that the CDH problem is hard. For any polynomial-time algorithm $\mathscr{A}$, under the random oracle model,*

$$\mathsf{Adv}^{\mathsf{link}}(\mathscr{A}) \leq \mathsf{negl}(\lambda)$$

where λ is the security parameter.

Proof. If there is an adversary that can break the anonymity game with probability ε, we can construct a simulator which breaks CDH assumption with probability $\frac{\varepsilon}{\mathrm{poly}(N,T,q_o)}$, where N denotes the total number of users, T denotes the total number of epochs, and q_o denotes the number of random oracle queries.

Revealing hashes instead of Bloom filter In the challenge stage, the P_{i^*}'s Bloom filter will have m elements. Instead of announcing the Bloom filter, we assume for the proof that users simply broadcast the outcomes of the hash functions used to construct the Bloom filter. This will only reveal more information to the adversary – so as long as we can prove the security when these hashes are revealed, we immediately guarantee security when the Bloom filter instead of the hash values are revealed.

Real-or-random version and sequence of hybrid games Instead of proving the left-or-right version of the game as in the security definition, we prove the real-or-random version. Namely, the adversary specifies one user P_i (instead of two) in the anonymity challenge (who must remain uncompromised at the end of the game), as well as a subset of P_i's previous encounters (which must remain uncompromised at the end of the game). The challenger flips a random coin, and either returns the faithful hash values to the adversary, or returns a list of random values from an appropriate range. The adversary's job is to distinguish which case it is.

We use a sequence of hybrid games. In the k-th game, replace the k-th hash (out of m hashes) in the challenge stage with some random value from an appropriate range.

Simulator construction The simulator obtains a CDH instance g^{α}, g^{β}. The simulator guesses that the k-th encounter in the anonymity challenge took place between users P_{i^*} and $P_{\widehat{i^*}}$ in time step τ. If the guess turns out to be wrong later, the simulator simply aborts. The simulator answers the following queries:

Expose handshake beacons. First, the simulator generates the DH beacons as below: except for users P_{i^*} and $P_{\widehat{i^*}}$ in time step τ, the simulator generates all other DH beacons normally. For P_{i^*} and $P_{\widehat{i^*}}$ in time step τ, their DH beacons will incorporate g^{α} and g^{β} respectively. Notice that the simulator does not know α, β, or the $\mathsf{dhk} := g^{\alpha\beta}$. Except for $g^{\alpha\beta}$, the simulator can compute all other dhks between two uncompromised users (even when one of g^{α} or g^{β} is involved) since the simulator knows the exponent of at least one DH beacon.

In generating the hashes for the Bloom filter, each hash can correspond to an encounter of the following types:

- Case 1: The hash does not involve an encounter in time τ. The simulator can compute the dhk and link identifier normally in this case.
- Case 2: The hash corresponds to an encounter in time τ, but at least one of the parties in the encounter is an uncompromised user (at the time of the challenge query) other than P_{i^*}, $P_{\widehat{i^*}}$. Notice that the simulator can compute the dhk (and hence the link identifier) in this case, since the simulator knows the exponent of the DH beacon of the other party.
- Case 3: The hash corresponds to an encounter in time τ, and between P_{i^*} and $P_{\widehat{i^*}}$. In this case, the simulator does not know the $\mathsf{dhk} = g^{\alpha\beta}$.
- Case 4: The hash corresponds to an encounter in time τ, and between P_{i^*} (or $P_{\widehat{i^*}}$) and the adversary. Suppose in this encounter in question, the adversary sent P_{i^*} the DH beacon g^{γ}. (The case for $P_{\widehat{i^*}}$ is similar and omitted). The simulator does not know the $\mathsf{dhk} = g^{\alpha\gamma}$ in this case.

Regardless of which type of encounter the hash corresponds to, as long as the simulator knows the dhk of this encounter, it can compute the link identifier and Bloom filters. Below, when we explain how to answer queries of the types "Handshake - Uncompromised users" and "Handshake - Adversary", we will explain how the simulator generates and records a link identifier for each of these encounters – even when it may not know the dhk (Cases 3 and 4). In this way, the simulator can answer queries for Cases 3 and 4 as well.

Handshake - Uncompromised users. Except for the encounter between P_{i^*} and $P_{\widehat{i^*}}$ in time epoch τ, for all other encounters, the simulator can compute the resulting dhks for both uncompromised users – even when one of g^{α} or g^{β} is involved. Therefore, the simulator computes and saves the dhk, which may later be used in answering "expose handshake beacon" queries.

For the encounter between P_{i^*} and $P_{\widehat{i^*}}$ in time τ, since the simulator does not know $\mathsf{dhk} := g^{\alpha\beta}$, it simply chooses a random link identifier and saves it internally, which will later be

used in answer "Expose handshake beacon" queries to construct Bloom filters.

Handshake - Adversary. Except when time step τ and user P_{i^*} or $P_{\widehat{i^*}}$ are involved, the simulator can proceed normally, and generate and dhk and other secrets that are derived as hashes of the dhk.

For time step τ, and P_{i^*} or $P_{\widehat{i^*}}$, something special needs to be done. Assume the adversary sends P_{i^*} handshake beacon g^γ (the case for $P_{\widehat{i^*}}$ is similar and omitted). The simulator does not know α or γ, hence it cannot compute the corresponding $\mathsf{dhk} := g^{\alpha\gamma}$. Without loss of generality, assume $g^\alpha < g^\gamma$. The simulator picks a random link identifier L^* – intended to be the link identifier for this encounter with the adversary. The simulator saves L^*, which will later be used in answering "Expose handshake beacon" queries.

The simulator informs the random hash oracle of the tuple $(\mathsf{L}^*, g^\alpha, g^\gamma)$. Later, random oracle may receive multiple queries of the form $H_0(g^\alpha||g^\gamma||Z)$. Suppose there are at most q_o of these queries. With probability $\frac{1}{q_o+1}$, the hash oracle never uses encK^* as the answer. With probability $1-\frac{1}{q_o+1}$, the hash oracle guesses one of these queries at random, and uses L^* as the answer. The simulator guesses correctly with probability at least $\frac{1}{q_o+1}$ where q_o is the number of hash oracle queries.

Compromise - Encounter. The adversary specifies a reference to a previous encounter (i, j, t'), where users P_i and P_j are uncompromised thus far. If i and j are not i^* or $\widehat{i^*}$, or $t' \neq \tau$, the simulator answers the query normally.

If $t' = \tau$, i and j cannot simultaneously be i^* and $\widehat{i^*}$, otherwise the simulator would have aborted. If one of i or j is i^* or $\widehat{i^*}$, the simulator can still answer the query, even without knowing α or β – since the simulator knows the exponent of the other player's DH beacon.

Compromise - User. If the adversary issues this query for user P_{i^*} or $P_{\widehat{i^*}}$, the simulator simply aborts. For all other uses, the query can be answered normally.

Random oracle. Above, we mentioned how the random oracle handles queries of the form $H_0(g^\alpha||g^\gamma||Z)$, where g^γ was a DH beacon from the adversary in a "Handshake - Adversary" query. For all other random oracle queries, the simulator picks random numbers to answer. The simulator records previous random oracle queries, so in case of a duplicate query, the same answer is given. Whenever the simulator needs to evaluate the hash function, it also queries its own random oracle.

Challenge - Anonymity. The Bloom filter hash values requested in the challenge stage must not have been queried in an "Expose handshake beacon" query. In the k-th hybrid game, the simulator outputs random values for the first k hashes. For the rest, the simulator constructs the answers normally – since these encounters happened before time τ, the simulator can compute their link identifiers and compute these hashes normally.

Without loss of generality, assume that $g^\alpha < g^\beta$. In the above simulation, the simulator makes all guesses correctly with probability at least $\frac{1}{\mathrm{poly}(N,T,q_o)}$. Conditioned on the fact that the simulator made all guesses correctly, unless the adversary queried $H_0(g^\alpha||g^\beta||g^{\alpha\beta})$, the $(k-1)$-th and k-th hybrid games are information theoretically indistinguishable from each other to the adversary. Now the adversary cannot have queried at any point $H_0(g^\alpha||g^\beta||g^{\alpha\beta})$ with more than negligible probability, since otherwise we can construct a simulator that outputs $g^{\alpha\beta}$ with non-negligible probability, thus breaking the CDH assumption.

□

Theorem 2 (Confidentiality)**.** *Assume that the CDH problem is hard. For any PPT algorithm $\mathscr{A}$, under the random oracle model,*

$$\mathsf{Adv}^{\mathsf{conf}}(\mathscr{A}) \leq \mathsf{negl}(\lambda)$$

where λ is the security parameter.

Proof. The simulator guesses that the adversary will issue a confidentiality between users P_{i^*} and $P_{\widehat{i^*}}$ in time epoch τ. If the guess turns out to be wrong later, the simulator simply aborts.

Suppose that simulator gets a CDH instance (g^α, g^β). The simulator would then answer all queries exactly as in the proof of Theorem 1, except for the challenge – instead of submitting a anonymity challenge, the adversary now submits a confidentiality challenge:

Challenge - Confidentiality. If i, j, and current time epoch τ does not agree with what the simulator had guessed, the simulator simply aborts. Otherwise, the simulator would have chosen a random link identifier in a "Handshake - Uncompromised users" query for $(i^*, \widehat{i^*}, \tau)$. The encounter key of this session is obtained by making a random oracle query on $H_2(L)$.

The simulator makes all guesses correctly with probability at least $\frac{1}{\mathrm{poly}(N,T,q_o)}$. Conditioned on the fact that all guesses are correct, the encounter key returned in the challenge stage is information theoretically indistinguishable from random, unless the adversary has queried $H_0(g^\alpha||g^\beta||g^{\alpha\beta})$ (assuming $g^\alpha < g^\beta$ without loss of generality). However, if the adversary makes such a random oracle query with non-negligible probability, we can construct a simulation that leverages the adversary to break the CDH assumption.

□

A.4 Co-Linking

Proposition 1. *Any non-interactive handshake protocol must be subject to a co-linking attack.*

Proof. In an non-interactive protocol, a user Alice publishes a message M in a certain time epoch. Suppose Bob and Charles have met Alice before (in encounters with link-ids L and L' respectively), and Alice has granted both of them permission to link her. Bob should be able to derive from his secret state and the message M, the link identifier L linking this encounter to the previous encounter L. Similarly, with his secret state and the message M, Charles should also be able to derive L'. Now trivially, if Bob and Charles collude, they can decide that the message M can be linked to previous encounters L and L'. □

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM

Caroline Tice, *Google, Inc.*
Tom Roeder, *Google, Inc.*
Peter Collingbourne, *Google, Inc.*
Stephen Checkoway, *Johns Hopkins University*
Úlfar Erlingsson, *Google, Inc.*
Luis Lozano, *Google, Inc.*
Geoff Pike, *Google, Inc.*

Abstract

Constraining dynamic control transfers is a common technique for mitigating software vulnerabilities. This defense has been widely and successfully used to protect return addresses and stack data; hence, current attacks instead typically corrupt vtable and function pointers to subvert a forward edge (an indirect jump or call) in the control-flow graph. Forward edges can be protected using Control-Flow Integrity (CFI) but, to date, CFI implementations have been research prototypes, based on impractical assumptions or ad hoc, heuristic techniques. To be widely adoptable, CFI mechanisms must be integrated into production compilers and be compatible with software-engineering aspects such as incremental compilation and dynamic libraries.

This paper presents implementations of fine-grained, forward-edge CFI enforcement and analysis for GCC and LLVM that meet the above requirements. An analysis and evaluation of the security, performance, and resource consumption of these mechanisms applied to the SPEC CPU2006 benchmarks and common benchmarks for the Chromium web browser show the practicality of our approach: these fine-grained CFI mechanisms have significantly lower overhead than recent academic CFI prototypes. Implementing CFI in industrial compiler frameworks has also led to insights into design tradeoffs and practical challenges, such as dynamic loading.

1 Introduction

The computer security research community has developed several widely-adopted techniques that successfully protect return addresses and other critical stack data [13, 20]. So, in recent years, attackers have changed their focus to non-stack-based exploits. Taking advantage of heap-based memory corruption bugs can allow an attacker to overwrite a function-pointer value, so that arbitrary machine code gets executed when that value is used in an indirect function call [6]. Such exploits are referred to as *forward-edge* attacks, as they change forward edges in the program's control-flow graph (CFG).

To make these attacks more concrete, consider a C++ program that makes virtual calls and has a use-after-free bug involving some object. After the object is freed, an attacker can reallocate the memory formerly occupied by the object, overwriting its vtable pointer. Later virtual calls through this object get the attacker's vtable pointer and jump to a function from the attacker's vtable. Such exploits are becoming commonplace, especially for web browsers where the attacker can partially control executed JavaScript code [14, 23].

Control-Flow Integrity (CFI) [1] guards against these control-flow attacks by verifying that indirect control-flow instructions target only functions in the program's CFG. However, although CFI was first developed over a decade ago, practical CFI enforcement has not yet been adopted by mainstream compilers. Instead, CFI implementations to date are either ad-hoc mechanisms, such as heuristic-driven, custom binary rewriting frameworks, or experimental, academic prototypes based on simplifying assumptions that prevent their use in production compilers [1, 9, 12, 29, 31–34].

In this paper, we present implementations of two mechanisms that provide forward-edge CFI protection, one in LLVM and one in GCC. We also provide a dynamic CFI analysis tool for LLVM which can help find forward-edge control-flow vulnerabilities. These CFI implementations are fully integrated into their respective compilers and were developed in collaboration with their open source communities. They do not restrict compiler optimizations, operation modes, or features, such as Position-Independent Code (PIC) or C++ exceptions. Nor do they restrict the execution environment of their output binaries, such as its use of dynamically-loaded libraries or Address Space Layout Randomization (ASLR).

The main contributions of this paper are:

- We present the first CFI implementations that are fully integrated into production compilers without restrictions or simplifying assumptions.
- We show that our CFI enforcement is practical and highly efficient by applying it to standard benchmarks and the Chromium web browser.
- We identify, discuss, and resolve the main challenges in the development of a real-world CFI implementation that is compatible with common software-engineering practices.

All our mechanisms verify targets of forward-edge in-

direct control transfers but at different levels of precision, depending on the type of target and the analysis applied. For example, C++ indirect-control transfers consist mostly of virtual calls, so one of our approaches focuses entirely on verifying calls through vtables. Our security analyses show that our CFI mechanisms protect from 95% to 99.8% of all indirect function calls. They are also highly efficient, with a performance penalty (after optimizations) ranging from 1% to 8.7%, as measured on the SPEC CPU2006 benchmark suite and on web browser benchmarks.

Most notably, our CFI mechanisms compare favorably to recent CFI research prototypes. The security guarantees of our work differ from these prototypes, but a comparison is nonetheless instructive, since our attack model is realistic, and our defenses give strong guarantees.

MIP [28] and CCFIR [34] state efficiency as their main innovation. In particular, on the SPEC Perl benchmarks (where they both were slowest), CCFIR reports 8.6% overhead, and MIP reports 14.9% to 31.3% overhead. Our mechanisms have a corresponding overhead of less than 2%, as we report in Section 7.2. For C++ benchmarks, which perform indirect calls more frequently, the performance differences can be even greater. Another recent CFI implementation, bin-CFI [35], reports overheads of 12% on the SPEC Perl benchmarks, but 45% for the C++ benchmark omnetpp, while the overhead is between -1% and 6.5% for omnetpp compiled using our mechanisms. Even the most recent CFI implementation, SAFEDISPATCH [19], must sacrifice software-engineering practicality and use profile-driven, whole-program optimization to achieve overheads comparable to ours (roughly 2% for all three of their Chromium benchmarks).

2 Attacks and Compiler-based Defenses

Software is often vulnerable to attacks that aim to subvert program control flow in order to control the software's behavior and assume its privileges. Typically, successful attacks exploit software mistakes or vulnerabilities that allow corruption of low-level state.

To thwart these low-level attacks, modern compilers, operating systems, and runtime libraries protect software integrity using a variety of techniques, ranging from coarse-level memory protection, through address-space layout randomization, to the fine-grained type-safety guarantees of a high-level language. In particular, machine-code memory is commonly write protected, and thread execution stacks are protected by placing them at random, secret locations, and by checking that secret values (a.k.a. canary values) remain unmodified. This guards return addresses and other stack-based control data against unintended overwriting [7, 13].

As a result of such compiler-based defenses becoming widely used, corruption of the execution stack or machine code has become a far less common means of successful attack in well-maintained, carefully-written software such as web browsers [20]. However, there has been a corresponding increase in attacks that corrupt program-control data stored on the heap, such as C++ vtable pointers (inside objects) [14], or function pointers embedded in data structures; these attacks subvert indirect control transfers and are known as "return-to-libc" or return-oriented programming [3, 8, 22, 25–27, 30].

In this paper we present three compiler-based mechanisms for further protecting the integrity of program control data. Focusing on the integrity of control-transfer data stored on the heap, two of our mechanisms enforce forward-edge CFI by restricting the permitted function pointer targets and vtables at indirect call sites to a set that the compiler, linker, and runtime have determined to be possibly valid. The third mechanism is a runtime analysis tool designed to catch CFI violations early in the software development life-cycle. Our mechanisms are efficient and practical and have been implemented as components of the GCC and LLVM production compiler toolchains. While they differ in their details, and in their security — such as in how precisely the program's CFG is enforced — all three of our implementations:

- add new, read-only metadata to compilation modules to represent aspects of the program's static CFG;
- add machine code for fast integrity checks before indirect forward-edge, control-flow instructions;
- optionally divert execution to code that performs slower integrity checks in certain complex cases;
- call out to error handlers in the case of failures; and
- may employ runtime library routines to handle important exceptional events such as dynamic loading.

Like all defenses, we aim to prevent certain threats and not others, according to an attack model. As in the original work on CFI, our model pessimistically assumes that an adversary can arbitrarily perturb most writable program data at any time [1]. The program code, read-only data, and thread register state cannot be modified. While pessimistic, this attack model has stood the test of time, and is both conceptually simple and realistic.

Similar to recent independent CFI work done concurrently with ours [19], and motivated by attackers' increasing focus on heap-based exploits, our mechanisms protect only forward-edge control transfers. Our attack model does not contain many types of stack corruption, since, as stated previously, effective defenses against such corruption are already in common use. Thus, our choice of attack model differs from most earlier work on CFI, except for work on mechanisms like XFI [12], which place the stack outside of pointer-accessible memory.

In our attack model we also depart from most previous CFI work by choosing to trust the compiler toolchain. For the integration of general-purpose defenses in pro-

duction compilers, we find relying only on stand-alone verification of the final, output binaries to be impractical — although well-suited to custom compilers for specific scenarios, such as in Google's Native Client [17, 21]. While eliminating trust in the compiler is a laudable goal [24], doing so increases complexity, reduces portability, and prevents optimizations, while providing only uncertain benefits. In particular, we know of no exploits on previous compiler-based defenses that justify the software engineering costs of eliminating trust in the compiler.

By adopting the above attack model, our mechanisms are practical as well as efficient. While many experimental CFI mechanisms have been constructed and described in the literature, none have been able to efficiently provide strong, precise integrity guarantees with the full support that programmers demand from a production compiler. In particular, incremental compilation and dynamic libraries have remained primary challenges for CFI implementations, as has achieving low performance overheads; these challenges have only recently started to be addressed in experimental prototypes that enforce more coarse-grained CFI policies [28, 34]. However, CFI enforcement that is too coarse grained may provide only limited protection [4, 10, 15, 16] against modern attackers.

In summary, by focusing on forward-edge CFI, and fully integrating into compilers, our mechanisms can enforce fine-grained CFI at a runtime overhead that improves on that of the best previous work.

Related Work. Following the original 2005 work on CFI, later revised as Abadi et al. [1], there have been a number of implementations that have extended or built-upon CFI: XFI by Erlingsson et al. [12], BGI by Castro et al. [5], HyperSafe by Wang and Jiang [31], CFI+Sandboxing by Zeng et al. [32], MoCFI by Davi et al. [9], CCFIR by Zhang et al. [34], Strato by Zeng et al. [33], bin-CFI by Zhang et al. [35], MIP by Niu et al. [28], and SAFEDISPATCH by Jang et al. [19].

These CFI-based mechanisms vary widely in their goals, tradeoffs and implementation details. To achieve low overhead, many enforce only coarse-grained CFI, which may be a weak defense [15].

XFI, Strato, HyperSafe, and BGI use control-flow integrity primarily as a building block for higher-level functionality, such as enforcing software-based fault isolation (SFI), or fine-grained memory-access controls. Some, like XFI, focus on statically verifying untrusted binary modules, to establish that CFI will be correctly enforced during their execution, and thus that they can be used safely within different address spaces, such as the OS kernel.

Many implementations of CFI are based on binary rewriting. XFI and the original work on CFI used the sound, production-quality Windows binary rewriter, Vulcan [11], as well as debug information in PDB files. These implementations construct precise control-flow graphs (CFGs) to ensure that all indirect control transfers are constrained in a sound manner. Other implementations — including MoCFI, CCFIR, and bin-CFI — are based on more ad hoc and fragile mechanisms. For example, MoCFI produces an imprecise CFG for ARM applications running on an iPhone based on runtime code dumping, disassembly, and heuristics. CCFIR relies on relocation tables and recursive disassembly with heuristics to identify code that needs to be protected. The code is then rewritten to use a special randomized springboard section through which all indirect control transfers happen. Bin-CFI also uses heuristic disassembly; however, unlike CCFIR, bin-CFI injects a CFI-protected copy of the text section into the original binary and uses dynamic binary translation to convert pointers between the two code copies at runtime.

Other implementations, including HyperSafe, MIP, CFI+Sandboxing, and SAFEDISPATCH are implemented as modifications to the compiler toolchain and compile source code to binaries with CFI protection. The first three are implemented as rewriting either assembly or the compiler's Intermediate Representation (IR) of machine code — essentially a more precise form of binary rewriting.

Our vtable verification (see Section 3) is most similar to SAFEDISPATCH: work done independently and concurrently with ours that adds passes to LLVM for instrumenting code with runtime CFI checks, and relies on profile-driven, whole-program optimization to reduce enforcement overhead. At a call site, SAFEDISPATCH checks that either (1) the vtable pointer is to a valid vtable, for the call site, or (2) the address in the vtable points to a valid method for the call site. However, SAFEDISPATCH disallows separate compilation, dynamic libraries, etc., and relies on profile-driven, whole-program optimization, which is not very practical.

The overhead of all these various CFI mechanisms ranges from 6% to 200%, with some significant variations. In particular, the recent papers on MIP, CCFIR, and Strato state their low CFI enforcement overhead as a main contribution; Strato also highlights its support for compiler optimizations. However, as mentioned previously, their overheads on comparable benchmarks are several times larger than those of our CFI mechanisms — and they are likely to perform even worse on C++ benchmarks. This is due in part to the different properties our work enforces: we limit our scope to protecting control data that is not already well-protected by other mechanisms.

3 VTV: Virtual-Table Verification

Vtable Verification (VTV) is a CFI transformation implemented in GCC 4.9 for C++ programs. VTV protects only virtual calls and does not attempt to verify other types of

indirect control flow. However, most indirect calls in C++ are virtual calls (e.g., 91.8% in Chrome), making them attractive targets for attackers. VTV is our most precise CFI approach: it guarantees that the vtable used by each protected virtual call is both valid for the program and also correct for the call site.

3.1 Problem Description

Virtual calls are made through objects, which are instances of a particular class, where one class (e.g., `rectangle`) inherits from another class (e.g., `shape`), and both classes can define the same function (e.g., `draw()`), and declare it to be virtual. Any class that has a virtual function is known as a *polymorphic class*. Such a class has an associated virtual function table (*vtable*), which contains pointers to the code for all the virtual functions for the class. During execution, a pointer to an object declared to have the type of a parent class (its *static type*, e.g., `shape`) may actually point to an object of one of the child classes (its *dynamic type*, e.g., `rectangle`). At runtime, the object contains a pointer (the *vtable pointer*) to the appropriate vtable for its dynamic type. When it makes a call to a virtual function, the vtable pointer in the object is dereferenced to find the vtable, then the offset appropriate for the function is used to find the correct function pointer within the vtable, and that pointer is used for the actual indirect call. Though somewhat simplified, this explanation is generally accurate.

The vtables themselves are placed in read-only memory, so they cannot be easily attacked. However, the objects making the calls are allocated on the heap. An attacker can make use of existing errors in the program, such as use-after-free, to overwrite the vtable pointer in the object and make it point to a vtable created by the attacker. The next time a virtual call is made through the object, it uses the attacker's vtable and executes the attacker's code.

3.2 Overview of VTV

To prevent attacks that hijack virtual calls through bogus vtables, VTV verifies the validity, at each call site, of the vtable pointer being used for the virtual call, before allowing the call to execute. In particular, it verifies that the vtable pointer about to be used is correct for the call site, i.e., that it points either to the vtable for the static type of the object, or to a vtable for one of its descendant classes. VTV does this by rewriting the IR code for making the virtual call: a verification call is inserted after getting the vtable pointer value out of the object (ensuring the value cannot be attacked between its verification and its use) and before dereferencing the vtable pointer. The compiler passes to the verifier function the vtable pointer from the object and the set of valid vtable pointers for the call site. If the pointer from the object is in the valid set, then it gets returned and used. Otherwise, the verification function calls a failure function, which normally reports an error and aborts execution immediately.

3.3 More Details

VTV differs from all previous compiler-based CFI implementations in that it allows incremental compilation rather than requiring that all files be recompiled if a single one is changed. VTV also does not forbid or restrict dynamic library loading. Such requirements and restrictions are common in research prototypes but impractical for real world systems.

Because VTV allows incremental compilation and dynamic loading, it must assume that its knowledge of the class hierarchy is incomplete during any particular compilation. Therefore, VTV has two pieces: the main compiler part, and a runtime library (*libvtv*), both of which are part of GCC. In addition to inserting verification calls at each call site, the compiler collects class hierarchy and vtable information during compilation, and uses it to generate function calls into libvtv, which will (at runtime) build the complete sets of valid vtable pointers for each polymorphic class in the program.

To keep track of static types of objects and to find sets of vtable pointers, VTV creates a special set of variables called *vtable-map variables*, one for each polymorphic class. At runtime, a vtable-map variable will point to the set of valid vtable pointers for its associated class. When VTV inserts a verification call, it passes in the appropriate vtable-map variable for the static type of the object, which points to the set to use for verification.

Because our vtable-pointer sets need to be built before any virtual calls execute, VTV creates special constructor init functions and gives them a high priority. The compiler inserts into these special functions the calls for building vtable-pointer sets. These functions run before standard initialization functions, which run before `main`, ensuring the data is in place before any virtual calls are made.

Vtable-map variables and vtable-pointer sets need to be read only to avoid introducing new vectors for attack. However, they must be writable when they are first initialized and whenever a dynamic library is loaded, since the dynamic library may need to add to the vtable-pointer sets. So, we need to be able to find all our data quickly. To keep track of the vtable-pointer sets, we wrote our own memory allocation scheme based on mmap. VTV uses this scheme when creating the sets; this lets it find all such sets in memory.

To find vtable-map variables, VTV writes them into a special named section in the executable, which is page-aligned and padded with a page-sized amount of zeros to prevent any other data from residing on the same pages. Before updating its data, VTV finds all memory pages that contain vtable-map variables and vtable-pointer sets and makes them writable. When it finishes the update, it finds

all appropriate pages and makes them read-only. Thus, the only times these VTV data structures are vulnerable to attack are during program initialization, and possibly during `dlopen` calls. The VTV code that updates our data structures uses pthread mutexes to prevent races between multiple threads.

3.4 Practical Experience with VTV

We encountered some challenges while developing VTV.

Declaring global variables. Originally, VTV declared vtable-map variables as COMDAT and as having global visibility, because incremental compilation can result in multiple compilation units defining the same vtable-map variable, and COMDAT sections can be coalesced by the linker. However, this did not work reliably because there are many ways in which programmers can override the visibility of a symbol: linker version scripts can explicitly declare which symbols have global visibility; the rest of the symbols become hidden by default. Also, if global symbols are not added to the dynamic symbol table, then vtable-map variables might not be global. Finally, calls to `dlopen` with the RTLD_LOCAL flag have similar effects. We found all these techniques at work in Chromium.

When some vtable-map variables do not have global visibility, there can be multiple live instances of a vtable-map variable for a particular class, each pointing to a different vtable-pointer set containing only part of the full vtable-pointer set for the class. If the verification function is passed a variable pointing to the wrong part of the set, execution aborts incorrectly. We call this the *split-set problem*.

We finally concluded that there is no existing mechanism for the compiler to ensure a symbol will always be globally visible. The only way to eliminate the split-set problem was to accept that there would be multiple live versions of some vtable-map variables. To handle the consequences of this new assumption, VTV keeps track of the first instance of a vtable-map variable for each class. When initializing any vtable-map variable, it first checks if it has already seen a version of that variable. If not, then it allocates a vtable-pointer set for the variable, makes the variable point to the vtable-pointer set, and registers the variable in its variable registry. All subsequent vtable-map variables for that class are then initialized to point to the same vtable-pointer set as the first one.

Mixing verified and non-verified code. VTV causes execution to halt for one of three reasons: (1) a vtable pointer has been corrupted; (2) the C++ code contains an incorrect cast between two class types (programmer error); or (3) the set of valid vtable pointers used for verification is incomplete. The split-set problem is an example of the last case. This can also occur if some files that define or extend classes are instrumented with vtable verification, and other files that define or extend part of the class hierarchy are not. A similar effect also can occur with libraries or plugins that pass objects in and out, if one is instrumented and the other is not.

lib.cc

```
#include "lib.h"
struct Derived_Priv : public Base {
  virtual ~Derived_Priv() {}
};

Base *GetPrivate() {
  return new Derived_Priv;
}
void Destroy(Base *pb) {
  delete pb; // virtual call #1
}
```

main.cc

```
#include "lib.h"
struct Derived : public Base {
        virtual ~Derived() {}
};

int main() {
  Derived *d = new Derived;
  Destroy(d);
  Base *pp = GetPrivate();
  delete pp; // virtual call #2
}
```

main.cc w/ VTV	lib.cc w/ VTV	vcall #1 OK	vcall #2 OK	Missing from vtable pointer set
Yes	Yes	Yes	Yes	Nothing
No	No	Yes	Yes	Nothing
Yes	No	Yes	No	`Derived_Priv`
No	Yes	No	Yes	`Derived`

Figure 1: Example of problems resulting from mixing verified and unverified code. If only main.cc is compiled with verification, the vtable pointer for `Derived_Priv` does not get added to the valid set for Base, so virtual call #2 fails to verify. If only lib.cc is compiled with verification, the vtable pointer for `Derived` does not get added to the valid set for Base, so virtual call #1 fails.

Figure 1 shows this problem: a header file, lib.h, declares a base class `Base`. The class `Base` contains one virtual function, its destructor. There are two source files, lib.cc and main.cc, that each includes lib.h and contains classes that inherit from `Base`, as shown in the upper part of Figure 1. The table in the lower part of Figure 1 shows the effects on the two marked virtual calls of compiling lib.cc and main.cc with and without VTV. Note that the only cases where both virtual calls pass verification are when everything is built with VTV or nothing is.

We encountered this problem in ChromeOS with the Chrome browser. There are two third-party libraries which are built without VTV and are distributed to ChromeOS as stripped, obfuscated binaries (these binaries are not part of the open-source Chromium project). To make matters worse, when we built the rest of Chrome

and ChromeOS with VTV and ran tests that exercised those libraries, we encountered verification failures.

To deal with the mixed-code problem in general, VTV was designed and written with a replaceable failure function. This function gets called if the verification function fails to find a vtable pointer in a valid set. To replace the default failure function, a programmer writes a replacement function (using the same signature, provided in a header file in libvtv), compiles it, and links it into the final binary. For Chrome we replaced the default failure function with a whitelist failure function. The whitelist function maintains an array with one record for each whitelisted library. The record contains the memory address range where the readonly data section for the library (which contains all of the library's vtables) is loaded. If a vtable pointer fails normal verification, it gets passed to the whitelist failure function. The function goes through the array, checking to see if the pointer is in any of the address ranges. If so, it assumes the pointer is valid and execution continues (verification succeeded). Otherwise, it reports an error and aborts.

Because the obfuscated libraries are dynamically loaded, the whitelist array records do not initially contain any addresses. If any records are empty when the whitelist failure function is called, then the function checks to see if the corresponding library has been loaded, and if so, it fills in the addresses before verification. For ChromeOS, our whitelist consists of the two third-party libraries mentioned above. This secondary verification, while not as accurate as normal VTV verification, still severely limits what an attacker can do, and with it we were able to execute all our tests on Chrome with no verification failures. Our secondary failure function only gets called in those cases where the main verification function fails. In that case it usually performs at most one alignment check and four pointer comparisons. Therefore, its overall impact on performance is small.

3.5 Alternatives & Enhancements for VTV

Since our performance overhead is reasonably good (ranging from 2.0% to 8.7% in the worst case, as we discuss in Section 7.1), we have not spent much time improving the performance of VTV. However, there are some things that could be done to improve these numbers. For various reasons, Chrome/ChromeOS currently cannot be compiled with devirtualization[1] enabled; we could enable devirtualization in Chrome and tune the devirtualizer to be more aggressive when combined with VTV. Partial inlining of the verification call sequences is another avenue we could explore, since call overhead accounts for a significant portion of our overall performance penalties. We could also implement secure methods for caching and reusing frequently verified values.

When we started implementing VTV, we decided that we did not want to modify any element of the toolchain except the compiler, especially because GCC can be used with a variety of different assemblers and linkers, and we did not want to modify all of them. An alternative approach would have been to have the compiler store the vtable-pointer sets as data in the assembly files. This data would be passed through the assembler to the linker. At link time the linker would see the whole program and could efficiently combine the vtable-pointer sets from the various object files into the appropriate final vtable-pointer sets. The dynamic loader, when loading the program, could load the pointers into our data sets and mark them read-only. This approach would eliminate ordering issues between functions that build vtable-pointer sets and functions that make virtual calls. The dynamic loader would also need to update the data, as appropriate, whenever it loaded a dynamic library that contained additional vtable pointers. A disadvantage of this alternative approach is that instead of requiring modifications only to the compiler, it would modify the entire toolchain: the compiler, the assembler, the linker, and the dynamic loader.

4 IFCC: Indirect Function-Call Checks

Indirect Function-Call Checks (IFCC) is a CFI transformation implemented over LLVM 3.4. It operates on LLVM IR during link-time optimization (LTO). IFCC does not depend on the details of C++ or other high-level languages; instead, it protects indirect calls by generating *jump tables* for indirect-call targets and adding code at indirect-call sites to transform function pointers, ensuring that they point to a jump-table entry. Any function pointer that does not point into the appropriate table is considered a CFI violation and will be forced into the right table by IFCC. IFCC collects function-pointers into jump tables based on function-pointer sets, like VTV's vtable-pointer sets, with one table per set.

IFCC forces all indirect-calls to go through its jump tables. This significantly reduces the set of possible indirect-call targets, and severely limits attacker options, preventing attacks that do not jump to a function entry point of the right type.

Each entry in a jump table consists solely of an aligned jump instruction to a function. The table is written to the read-only text area of the executable and is padded with trap instructions to a power-of-two size so that any aligned jump to the padding will cause the program to crash. Since the size of the table is a power of two, IFCC can compute a mask that can be used at call sites to force

[1] Devirtualization is an optimization that replaces virtual calls with a fast-path/slow-path mechanism: the fast path uses a direct call to the most common target, with a conditional check to make sure this is right; the slow path falls back on the normal virtual call mechanism. Devirtualization reduces the number of indirect calls and verifications and improves the overall performance of VTV.

a function pointer into the right function-pointer set. For example, if each jump-table entry takes up 8 bytes, and the table is 512 bytes in size, then there are 64 entries in the table, and the mask would be 111111000 in binary, which is 504 in decimal.

IFCC rewrites IR for functions that have their address taken; we call these *address-taken* functions. The main transformation replaces the address of each such function with the address of the corresponding entry in a jump table. Additionally, indirect calls are replaced with a sequence of instructions that use a mask and a base address to check the function pointer against the function-pointer set corresponding to the call site. The simplest transformation subtracts the base address from the pointer, masks the result, and adds the masked value back to the base. If the pointer was in the right function-pointer set before this transformation, then it remains unchanged. If not, then a call through the resulting pointer is still guaranteed to transfer control only to addresses in the function-pointer set or to trap instructions. Note that every pointer in a jump table is a valid function pointer (although some of them immediately hit trap instructions when they are called), so they can correctly be passed to external code.

IFCC can support many kinds of function-pointer sets, each with different levels of precision. For example, the most precise version would have one function-pointer set per function type signature. However, real world code does not always respect the type of function pointers, so this can fail for function-pointer casts. We will focus on two simple ways of constructing function-pointer sets: (1) *Single* puts all the functions into a single set; and (2) *Arity* assigns functions to a set according to the number of arguments passed to the indirect call at the call site (ignoring the return value).

Note that although we implemented only two simple types of tables, any disjoint partitioning of the function-pointer types in the program will work, as long as each call site can be uniquely associated with a single table. This is true no matter what analysis is used to generate these tables — be it static, dynamic, or manual. So, for example, the compiler could perform a detailed analysis of escaping pointers and use it to separate these pointers into their own tables.

The following example demonstrates the IFCC technique for a simple program. Consider a function `int f(char a) { return (int)a; }` and a main function that makes an indirect call to `f` through a function-pointer variable `g`. In LLVM IR, the symbol `@f` will refer to function defined above; IFCC adds a `@baseptr` symbol that stores a pointer to the first function pointer in the generated jump table. Before IFCC, the LLVM IR contains an indirect call instruction `%call = call i32 %2(i8 signext 0)` to the function pointer stored in variable `%2`.

IFCC generates a new symbol `@f_JT` and defines it in the IR as an external function. It finds each instance where the program uses the address of `@f` and makes it use the address of `@f_JT` instead. It also creates a jump table of the form:

```
  .align 8
  .globl f_JT
f_JT:
  jmp f
```

This defines the symbol `@f_JT` and satisfies the linker. IFCC instruments the code before the indirect call with instructions that transform the pointer. There are several ways to perform this transformation. We show two techniques, one that requires large alignments, and another for when large alignments are not supported. The requirement for large alignment in one scheme is because the base pointer must be aligned to the size of its table. This makes the base a prefix of each entry in its table.

When the object format and the kernel support large table alignments (e.g., greater than one page), IFCC can use a compact set of instructions to transform a pointer. The following IR assumes integer representations of `@baseptr` in `%1` and `@mask` in `%2`, and a pointer to `@f` in `%3`.

```
%4 = and i64 %2, %3
%5 = add i64 %1, %4
%6 = inttoptr i64 %5 to i32 (i8)*
%7 = call i32 %6(i8 signext 0)
```

In x86-64 assembly, this becomes:

```
and     $mask, %rax
add     $baseptr, %rax
callq   *%rax
```

The ELF format supports arbitrary alignments but the Linux kernel does not (as of version 3.2.5, under ASLR with Position-Independent Executables (PIE)). Under ASLR, the kernel treats the beginning of each ELF segment as an offset and generates a random base to add to the offset. The base is guaranteed to be aligned to a page boundary (2^{12}) but the resulting address is not guaranteed to have larger alignment.

Under these circumstances, IFCC changes the way it operates on function pointers; instead of adding a masked pointer to a base, it computes the difference between the base address and the function pointer, masks this value with the same mask as before, and adds the result to the base. This ends up generating 3 instructions for pointer manipulation in x86-64: a `sub`, then an `and`, then an `add`.

4.1 Practical Experience with IFCC

We modified the Chromium build scripts to build under LTO as much of the code as possible. It built 128 files as x86-64 ELF objects, and 11,012 files as LLVM IR. We

then applied, separately, the Single and Arity versions of IFCC in this configuration.

Like VTV, IFCC suffers from false positives due to external code. In particular, any function that was not defined or declared during link-time optimization will trigger a CFI violation. This can happen for several reasons. First, JIT code (like JavaScript engines) can generate functions dynamically. Second, some external functions (like `dlsym` or `sigaction`) can return external function pointers. Finally, some functions can be passed to external libraries and can be called there with external function pointers; this is common in graphics libraries like `gtk`. The number of false positives varies greatly depending on the external code, ranging from extremely frequent in the case of JIT-generated code to extremely infrequent in the case of signal handlers.

To handle function pointers returned by external code, we added a fixed-size set of special functions to the beginning of each table. These functions perform indirect jumps through function pointers stored in an array. IFCC rewrites all calls to external functions (including `dlsym`) that return function pointers and inserts a function call that takes the pointer and writes it to the array if it is not already present. It returns a pointer to the table entry that corresponds to the array entry used to store the pointer. If the array has no more space, then it halts the program with an error. This converts function pointers from external code into table entries at the expense of adding a small number of writable function pointers to the code. This memory can be protected using the techniques described in Section 3.4, though the prototype in this paper does not perform this protection.

To handle functions passed to external functions, IFCC must find all cases in which functions are passed to external code and must rewrite the functions to not test their function pointers against the jump tables generated by IFCC. We added a flag to the IFCC plugin that takes the name of a function to skip in rewriting. To discover these function pointers, we added a warning mode to the IFCC transformation that prints at run time the names of functions that make indirect calls to functions outside the function-pointer sets. We found 255 such functions in Chromium, mostly associated with graphics libraries.

4.2 Annotations

The version of IFCC described in this section provides automatic methods for discovering and handling false positives. To improve maintainability of software with IFCC, however, we have implemented a different version that uses annotations instead of compile-time flags and uses custom failure functions like VTV. This is the version that we are working on upstreaming into LLVM.

Instead of functions being forced into the appropriate jump table, they are checked using the same code sequences as above, and any pointer that fails the check is passed to a custom failure function. IFCC's default failure function prints out the name of the function in which the failure occurred, and the value of the pointer that failed the check. This version of IFCC adds a comparison, a jump, and function call to the inserted instruction sequence. However, it gives greater flexibility to the resulting code in handling false positives, as discussed for VTV.

This new implementation provides annotations and a simple interprocedural dataflow analysis to help detect and handle these problems automatically. We provide two annotations that programmers can add using attribute notation: `__attribute__((annotate()))`.

- `cfi-maybe-external` is applied to local variables/parameters as well as to pointers in data structures.
- `cfi-no-rewrite` is applied to functions.

The dataflow analysis in IFCC finds external function pointers and traces their flow into indirect calls and into store instructions. It also traces the flow from `cfi-maybe-external`-annotated pointers and other variables into indirect calls and store instructions. It produces compile-time warnings if it finds a store instruction for an external function pointer and the pointer in the store instruction did not come from a location annotated by `cfi-maybe-external`. The annotations then can be used as a kind of whitelist in the CFI failure function, or these indirect calls can be skipped in rewriting.

The annotation `cfi-no-rewrite` means that all indirect calls in the annotated function might use external function pointers. The information from this annotation can be used either to build a whitelist or to skip rewriting. Our implementation currently skips rewriting for these indirect calls.

These annotations are also useful for cases that IFCC cannot detect, like callbacks buried deep inside data structures passed to external code. Calls to these functions will generate CFI violations at run time; these violations are false positives, and the locations of these indirect calls can be annotated with, e.g., `cfi-maybe-external` to indicate this to the CFI failure function.

It might seem like all an adversary has to do is to find one of the locations that has been annotated with `cfi-maybe-external` and overwrite a pointer that flows into it, and this will defeat IFCC. However, these annotations merely convey information to the CFI failure function; this function can perform arbitrarily complex checks make sure that function pointers that violate CFI are still valid. For the purposes of our evaluation, we implemented a simple failure function, as described above.

5 FSan: Indirect-Call-Check Analysis

The more precise the control-flow graph used in a CFI implementation, the harder it becomes for an attacker to exploit a program while staying within CFI-enforced bounds: a more precise CFG leads to fewer choices in targets for indirect control-transfer instructions. However, building a precise CFG is a hard problem, and programming techniques like function-pointer type punning and runtime polymorphism exacerbate this problem. Every time a CFI mechanism faces uncertainty, it is forced to be conservative to preserve correctness. Although this strategy guarantees correctness, it may result in a loss of security. Thus, techniques that reduce uncertainty about the CFG can increase security. We can achieve the best practical results by combining knowledge of programming language constructs (such as vtables), static analysis (such as we do for IFCC), and dynamic analysis. To that end, we designed FSan—an optional indirect call checker—and integrated it into Clang, LLVM's front end.

Clang's undefined behavior sanitizer (UBSan) instruments C and C++ programs with checks to identify instances of undefined behavior. FSan was added to UBSan in LLVM 3.4. FSan detects CFI violations at runtime for indirect function calls.[2] FSan operates during the translation from the Clang AST to LLVM IR, so it has full access to type information, allowing it to make more accurate checks than IFCC, which uses IR alone.

FSan is a developer tool designed to perform optional type checking. In particular, it is not designed to defend against attacks. Instead, it is designed to be used by developers to identify CFI violations that may lead to security issues. As a fully accurate checker (it checks definedness exactly according to the definition in the C++ standard), it can also be used to help guide the development of control-flow integrity techniques by identifying properties of interest to be checked in the field.

FSan prefixes each function emitted by the compiler with 8 (on x86-32) or 12 (on x86-64) bytes of metadata. Table 1 shows the layout of these bytes; they are executable and cost little in performance, since the first two bytes encode a relative branch instruction which skips the rest of the metadata. The next two bytes encode the instructions `rex.RX push %rsp` (on x86-64) or `incl %esi ; pushl %esp` (on x86-32); this sequence of instructions is unlikely to appear at the start of a non-instrumented function body, and we observed no false positives in Chromium due to this choice of prefix.

Each indirect call site first loads the first four bytes from the function pointer, and compares it to the expected signature — the optionality of FSan arises from selecting a signature unlikely but permitted to appear at the start of an uninstrumented function. Because GCC at optimization level -O2 and higher and Clang at any optimization level will align functions to 16 bytes, this initial read succeeds for each function compiled with these compilers, regardless of the length of the function. This assumes GCC-compiled system libraries are compiled with -O2 or higher.

[2]The undefined behavior sanitizer also includes a vtable-pointer checker which is not described here.

Kind	Offset	Data	Interpretation
Signature	0 1	0xeb 0x06/0x0a	`jmp .+0x08/0x0c`
	2 3	0x46 0x54	'F' 'T'
RTTI	4	Pointer to `std::type_info` for the function's type (4/8 bytes)	

Table 1: Function prefix data layout for the optional function type checker.

If the signature matches, then the next 4 or 8 bytes are loaded and compared against the expected function Run-Time Type Information (RTTI) pointer, which is simply the RTTI pointer for the function type of the callee expression used at the call site. If the pointers are unequal, then a runtime function is called to print an appropriate error message. A pointer equality test is sufficient because the function RTTI pointer for a particular function type is normally unique. This is because the linker will normally (but not always) coalesce RTTI objects for the same type, as they have the same mangled name.

The condition for undefined behavior as specified by the C++ standard is that the function types do not match (see C++11 [18, expr.reinterpret.cast], "The effect of calling a function through a pointer to a function type [...] that is not the same as the type used in the definition of the function is undefined"), so FSan is precise (no false positives or false negatives) with respect to this paragraph of the standard when both the caller and callee are compiled with the checker (provided that the linker coalesces RTTI symbols). However, FSan has not been implemented for C, and indeed would not work in its present form, mainly because the C rules relating to the definedness of calls to functions without a prototype are more complex.

Note that the RTTI pointer for a vtable function call is less precise than the vtable-pointer set check in VTV. FSan checks that each function has the correct type but not whether it was in the original program's CFG.

5.1 Practical Experience with FSan

We evaluated FSan by applying it to Chromium: we ran an instrumented version of the main browser executable, and FSan produced a variety of undefined-behavior reports. Two of the main categories of reports we observed were:

- Template functions whose parameters are of a templated pointer type, which are cast to functions whose parameters are of void pointer type so that they can be used as untyped callbacks;

- Functions that take context parameters as typed pointers in the function declaration but void pointers or pointers to a base class at the call site.

The fix for these types of bugs is simple in principle: give the parameters a void pointer type and move the casts into the function body. One instance of each type of problem was found and fixed in the Skia graphics library that Chromium uses. This eliminated much of the low-hanging fruit reported by FSan; most of the remaining problems were more widespread in the codebase and thus will take more effort to deal with. For example, V8 uses callbacks to implement a feature known as "revivable objects" which Blink (née WebKit) relies on heavily; in many cases these callbacks were implemented using the derived types expected by the object implementation, rather than V8's base value type.

6 Security Analysis

In order to evaluate the efficacy of security techniques it is important to apply them to the real-world code they are expected to protect and measure the impact. To this end, we analyzed Chromium compiled with VTV and GCC, and with IFCC and Clang.

One consequence of implementing security mechanisms in the compiler is that it is important to evaluate the final output rather than simply the output of the particular compiler pass. The reasons for this are twofold: (1) later optimization passes may transform the security mechanism in unpredictable ways; and (2) the final linking step adds additional binary code to the final executable that the security pass never sees.

Basic optimizations such as common-subexpression elimination and loop-invariant code motion can eliminate redundant checks or hoist checks out of loops. Although such optimizations are generally acceptable from a correctness point of view, they may be impermissible from a security standpoint. For example, consider two consecutive calls to C++ member functions `obj.foo(); obj.bar();`. A security mechanism that protects virtual function calls, such as VTV, can load a vtable pointer into a callee-saved register, perform the verification, and then perform the two calls:

```
movq    (%r12), %rbx  ; set rbx to the vptr
movq    %rbx, %rdi
callq   verify_vtable ; verify_vtable(vptr)
movq    %r12, %rdi
callq   *16(%rbx)     ; obj.foo()
movq    %r12, %rdi
callq   *24(%rbx)     ; obj.bar()
```

This is perfectly correct behavior since the first function call is guaranteed by the platform ABI to preserve the value of the register. However, if `rbx` is spilled to the stack in `foo()` and is later overwritten, e.g., via a buffer overflow on the stack, then the call to `bar()` will be to an attacker-controlled location. An alternative to using a callee-saved register is to explicitly spill/reload the register to/from the stack, which has similar security concerns.

Since protecting the stack is outside the scope of this work, the compiler has significantly more freedom to eliminate this sort of redundant check.

Although it is difficult to meaningfully quantify the security provided by a mitigation measure, recent work by Zhang and Sekar [35, Definition 1] introduced the Average Indirect-target Reduction (AIR) metric

$$AIR = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \frac{T_i}{S}\right)$$

where n is the number of indirect control-transfer instructions (indirect calls, jumps, and returns), T_i is the number of instructions the ith indirect control transfer instruction could target after applying a CFI technique, and S is the size of the binary.

It's clear that for any reasonable CFI technique and a large binary, $T_i \ll S$ for all indirect control-transfer instructions transformed by the technique. Similarly, $T_i \approx S$ for all other indirect control-transfer instructions. So, AIR reduces to the fraction of indirect control transfer instructions that are protected by the technique.[3]

Since we are focused on protecting forward edges, we consider the related metric forward-edge AIR, or fAIR, which performs the same computation as AIR, but the average is taken only over the forward-edge indirect control transfer instructions: indirect calls and jumps.

To compute the statistics reported in the rest of this section, we modified LLVM's object-file disassembler to perform a hybrid recursive and linear scan through the Chromium binary, reconstructing functions and basic blocks on which we performed our analysis. This process was aided by ensuring that Chromium was compiled with debugging information including symbols (cf. Bao et al. [2]). This disassembler was used as part of a stand-alone tool to find all indirect control transfer instructions. For each such instruction, the tool walks backward through the CFG, looking for the specific protection mechanism. It also attempts to find constants which are inserted into registers used for the call or jump. See Table 2 for a break down of forward-edge indirect control transfer (fICT) instructions in Chromium.

VTV. Compiling a recent version of Chromium using GCC with vtable verification leads to a final binary containing 124,325 indirect calls and 18,453 indirect jumps for a total of 142,778 fICT instructions. Of these, 6,855 are neither constant nor protected by vtable verification, giving $fAIR_{\mathrm{VTV}} = 95.2\%$. The majority of the unprotected

[3] This demonstrates that AIR — and our related fAIR — is at best a weak proxy for measuring security. Unfortunately, an actual metric for the security a CFI technique provides has thus far remained elusive.

fICT	VTV	IFCC
Constant	7,410	5,957
Constant, spilled†	7,334	315
Protected	113,617	154,244
Protected, spilled†	7,562	33,914
Unprotected	6,855	908
Total	142,778	195,338

Table 2: Forward-edge indirect control transfer (fICT) instructions in Chromium. Their arguments may be placed in three classes: (a) a type of constant, (b) an indirect address protected by CFI, and (c) an unprotected address. Constant-argument instructions include indirect jumps in the PLT which target a read-only GOT section and indirect jump instructions implementing switch statements, as well as indirect call instructions with constant targets.
† The targets for these indirect control transfer instructions are either spilled to the stack explicitly or are in callee-saved registers which are potentially spilled by intervening function calls.

fICTs come from C libraries, function-pointer adapter classes, and C-style callbacks.

Although more than 89% of the protected or constant fICTs are used almost immediately after being verified or loaded from read-only memory, in 14,896 instances (about 10%), the indirect target is potentially spilled to the stack. But this is not a flaw in our protection (see below).

IFCC. Compiling the same version of Chromium using Clang with IFCC (Single) produces a different binary containing 175,396 indirect calls and 19,942 indirect jumps for a total of 195,338 fICT instructions. Having more calls and jumps is what we would expect since the link-time optimizer has more inlining opportunities than is the case when optimizing one translation unit at a time.[4]

Since IFCC is designed to protect all fICT instructions, not just C++ virtual member function calls, only 908 fICT instructions are left unprotected. This gives $fAIR_{\text{IFCC}} = 99.5\%$. In fact, this is an over estimate of the number of unprotected fICT instructions. Of the 908 unprotected instructions, 512 correspond to the special functions created for function pointers returned from non-instrumented functions. Discounting those gives the more accurate value of $fAIR_{\text{IFCC}} = 99.8\%$. Most of the remaining unprotected fICT instructions correspond to functions which are explicitly not instrumented. (See Section 4.1 for a discussion of both of these.) The remaining handful come from the C run time statically linked into every binary.

With IFCC, about 18% of the constant or protected fICTs have targets which are potentially spilled to the stack. However this is not a fatal flaw, as discussed immediately below, since we are assuming that the stack is protected by some other means.

Stack spilling implications. For the purpose of our techniques, spilling target values to the stack introduces no additional security risk, since an attacker who can overwrite one value on the stack can easily overwrite a saved return address. This does have serious implications for CFI schemes that attempt to protect backward edges.

Our experience shows the importance of verifying a protection mechanism's intended invariants on the final binary output after all optimizations, including architecture-dependent optimization in the compiler backend, have taken place and the language runtime has been linked in.

Counting ROP gadgets. It is common in CFI papers to count the number of return-oriented programming gadgets that remain after applying the protection mechanism. Since we are explicitly not protecting return instructions, it does not make sense to count gadgets.

7 Performance Measurements and Results

We measured the performance of our approaches both on the C++ tests from the SPEC CPU2006 benchmark suite and on the Chromium browser running Dromaeo, SunSpider, and Octane. Except where otherwise specified, the VTV tests were run all on an HP Z620 Xeon E52690 2.9GHz machine, running Ubuntu Linux 12.04.2, and the IFCC and FSan tests were run on an HP Z620 Xeon E5550 2.67GHz machine, running the same OS. We turned off turbo mode and ASLR on these machines, as doing so significantly reduced the variation in our results.

The Chromium web browser is a large, complex, real-world application, comprising over 15 million lines of C++ code in over 50,000 source files, and containing hundreds of thousands of virtual calls. It links in many third-party libraries and makes extensive use of dynamic library loading. It is also representative of the type of target attackers are interested in. For all these reasons, Chromium makes an excellent test for measuring the effects of our CFI approaches on real-world systems. Both VTV and IFCC were able to successfully build fully-functional, protected versions of Chromium.

7.1 VTV Performance

Since verification adds instructions to the execution, some performance penalty is unavoidable. Initially, we ran SPEC CPU2006 C++ benchmarks with and without VTV to get a baseline. Table 3 shows that omnetpp, astar, and xalancbmk suffer a noticeable performance penalty in this naive implementation, ranging from 2.4% to 19.2%. We improve on this later. The other four benchmarks (povray, namd, soplex, and dealII) showed no significant performance effects. To determine why, we collected statistics on those benchmarks, for both compile-time

[4] There is a corresponding decrease in the number of return instructions for the same reason.

Test	no VTV (seconds)	VTV (seconds)	% slowdown
omnetpp	320.70	346.42	8.0
astar	440.61	450.95	2.4
xalanc.	248.86	296.53	19.2
namd	445.19	445.32	*
dealII	344.89	348.39	*
soplex	235.20	236.46	*
povray	181.18	181.87	*

Table 3: Untuned SPEC run-time numbers, at -O2. The asterisks indicate changes that are too small to be of any significance. These numbers are the minimum out of three runs (standard deviation is very close to zero).

Test	virtual calls (static)	verified calls (dynamic)	run time (secs)	verified calls per second
namd	2	0	445.32	0
dealII	2,118	201,867,094	348.39	579,428
soplex	720	4,846,399	236.46	20,496
povray	159	159,186	181.87	875
omnetpp	1,312	1,029,110,532	346.42	2,970,702
astar	2	2,780,359,179	450.95	6,165,560
xalanc.	15,753	2,629,817,426	296.53	8,868,639
dromaeo	NA	6,705,708,649	2379.34	2,818,303
octane	NA	113,037,194	66.41	1,702,214
sunspi.	NA	27,068,246	16.36	1,654,943

Table 4: Verifications per second when running SPEC CPU2006 C++ benchmarks and Chrome with VTV.

(static) and run-time (dynamic) numbers of verified virtual calls. From this we calculated the number of verified calls per second. As shown at the top of Table 4, those tests do not perform enough calls per second to noticeably affect performance, so we did not include them in any further analyses of VTV.

Next, we looked at reducing VTV's performance penalty. To determine the minimum lower bound penalty, we considered two sources of performance overhead: (1) the cost of making the verification function calls; and (2) the (potential) cost due to the overall increase in code size (code bloat). We measured these by doing two experiments on the three SPEC benchmarks of interest. First, we replaced the bodies of the functions in libvtv with stubs. Second, we inserted an unreachable region of code preceded by an unconditional jump over the region just before each virtual call instruction (the unreachable region represented the code that would be inserted by VTV). Our results can be seen in Table 5. Note that making calls with stubs must increase the number of instructions, just as with the code bloat test. Therefore the stubs penalty automatically includes the code bloat penalty. This shows that

	Code Bloat % Slowdown	VTV Stubs % Slowdown
omnetpp	0.0	2.4
astar	0.2	1.0
xalancbmk	0.8	4.7

Table 5: Results of lower bound experiments for VTV.

even if we could reduce to zero the time spent executing inside the verification function, the minimum lower bound VTV penalty for these tests ranges from 1.0% to 4.7%. Note that the lower bound is test-specific and depends on the number of virtual calls a test executes.

We then tried various options to reduce VTV's performance penalties. The two most effective options were: using profile guided optimizations (PGO) to improve devirtualization, via GCC's -ripa flag, thus reducing the overall number of virtual calls; and statically linking libvtv itself, to reduce the level of indirection at each verification call. We re-ran the SPEC benchmarks using these various options, and the results are shown at the top of Figure 2. The xalancbmk test had the worst performance with VTV, so it is instructive to consider its results under optimization: devirtualization brought the performance penalty from 19.2% down to 10.8%, and static linking reduced it further to 8.7%. The lower bound of VTV is 4.7% for xalancbmk (see Table 5).

Chrome interacts with many system libraries, so to avoid the problems of mixing verified and unverified code with VTV we built and ran a verified version of Chrome in a verified version of ChromeOS on a Chromebook (thus building all the libraries with verification as well). We built ChromeOS version 28 with VTV, and ran the Dromaeo, SunSpider 1.0.2, and Octane 2.0 benchmarks with it. For these tests, we loaded our images onto a Chromebook with an Intel Celeron 867 chip, pinned at 1.3GHz, with ASLR turned off, and we ran the tests there, with and without VTV. The bottom of Figure 2 shows the performance costs. We were not able to build Chrome with PGO and devirtualization, nor with the statically linked libvtv, so for these measurements we only have the naive, untuned VTV numbers. For Octane we saw a 2.6% penalty with VTV. SunSpider had a 1.6% penalty. Dromaeo had a fair amount of variation across the full set of micro-benchmarks, but the overall performance penalty across all of them was 8.4%. We expect that adding devirtualization would significantly improve these numbers. As with the SPEC benchmarks, the performance penalty varies depending on the number of verified calls made at runtime. We measured this for each of these tests (see Table 4). As expected, Dromaeo, which had the largest penalty, makes significantly more verified calls/second than the other two.

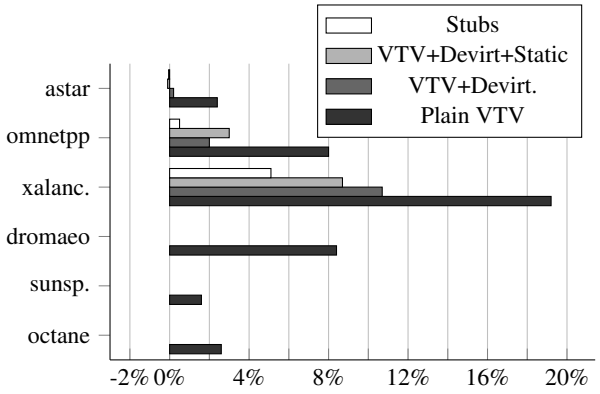


Figure 2: Relative performance overhead of VTV, with various tuning options, for the SPEC 2006 C++ benchmarks and for Chrome browser.

7.2 IFCC and FSan Performance

The performance of code compiled under IFCC depends on how often it makes indirect calls. IFCC adds code to every indirect call site that is not explicitly skipped by a command-line directive. The amount of code it adds depends on the version of IFCC: if tables are small enough to fit in a page in memory, then it can use the transformation that adds only two instructions (comprising 14 bytes) to each site. Otherwise, it uses the subtraction version, which adds 4 instructions (which become 20 bytes). Each indirect call has additional extra overhead from the `jmp` in the indirect call table(s); and jumps through a table might have effects on instruction-cache usage. Finally, when rewritten code receives a pointer from `dlsym` or from a dynamically-linked library, this pointer is wrapped using linear search through a fixed-length array; this is a slow operation but should not happen often.

The exact instructions added by FSan depend on the specific optimization level used, but we found that it usually adds about 12 instructions to each call site.

Figure 3 shows results from running C++ programs from the SPEC benchmark suite under IFCC and FSan and provides relative performance overhead compared to an optimized version compiled using Clang; each running time is the minimum of 10 executions. As expected, LTO outperforms both IFCC transformations in most cases. This is because IFCC adds instructions to the base LTO-compiled binary, and these instructions reduce performance of the executable. The cases in which IFCC outperforms LTO involve only small differences in performance and are likely due to effects similar to the noise discussed by Mytkowicz et al. [24], so we do not analyze them further here.

We ran the Dromaeo benchmark on Chromium 31.0.1650.41 built with Clang LTO, a version built with IFCC Single, and a version built with IFCC Arity. Single got 96.6% of the LTO score, and Arity got 96.1%; higher is better in Dromaeo, so this is about a 4% overhead, as shown in Figure 3a. We also built a version of Chromium and the SPEC CPU2006 benchmarks with the annotation version of IFCC, and we saw similar results.

IFCC had nearly the same performance as LTO for both Single and Arity versions of the SPEC CPU2006 benchmarks, except for xalancbmk. FSan had effects similar to IFCC. The xalancbmk benchmark suffers the most performance degradation from IFCC; this is expected due to it having the most dynamic virtual calls, as shown in Table 4. Similarly, Dromaeo has a large number of virtual calls and has the second highest overhead. So, as with VTV, the performance overhead is directly related to the number of indirect calls.

7.3 Comparison to Prior Work

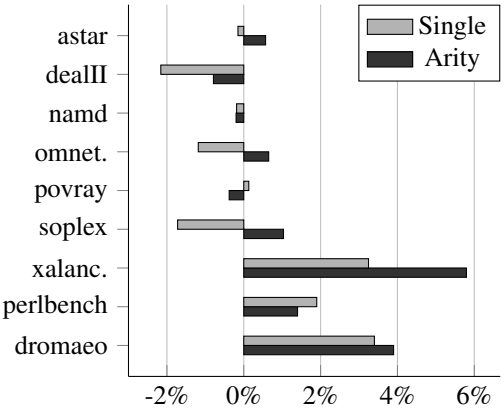
The SPEC Perl benchmark is worth highlighting. As Niu and Tan [28] point out, Perl is, in some sense, a worst case for CFI techniques for C — whereas C++ code can be even worse. Perl operates by translating the source code into bytecode, then sits in a tight loop, interpreting each instruction by making an indirect call. This worst case behavior is apparent in the performance of four recent CFI implementations: CCFIR by Zhang et al. [34], bin-CFI by Zhang and Sekar [35], Strato by Zeng et al. [33], and MIP by Niu and Tan [28]. The overheads reported for CCFIR, bin-CFI, Strato, and MIP are 8.6%, 12%, 15%–25%, and 14.9%–31.3%, respectively.

In contrast, our own work has less than 2% overhead (see Figure 3a). We are able to achieve this significant speed up over prior work by focusing only on forward edges as well as leveraging the compiler to apply optimizations. This gives different security guarantees, but we believe our attack model comports well with reality.

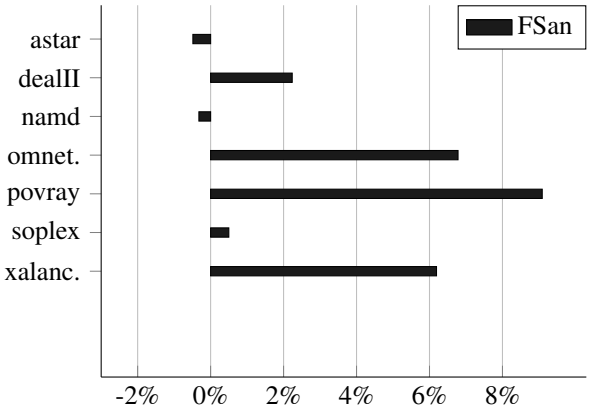
8 Conclusions

This paper advances the techniques of Control-Flow Integrity, moving them from research prototypes to being firmly in the domain of the practical. We have described two different principled, compiler-based CFI solutions for enforcing control-flow integrity for indirect jumps: vtable verification for virtual calls (VTV) guarantees that the vtable being used for a virtual call is not only a valid vtable for the program but is semantically correct for the call site; and indirect function-call checking (IFCC) guarantees that the target of an indirect call is one of the address-taken functions in the program. We also present FSan, an optional indirect call checking tool which verifies at runtime that the target of an indirect call has the correct function signature, based on the call site.

We have demonstrated that each of these approaches is feasible by implementing each one in a production compiler (GCC or LLVM). We have shown via security analysis that VTV and IFCC both maintain a very high level



(a) Relative overhead of IFCC enforcement (baseline LLVM LTO) for SPEC CPU2006 benchmarks and the Dromaeo benchmark.



(b) Relative overhead of the FSan optional indirect-call checking (baseline Clang) for the C++ benchmarks in SPEC CPU2006.

Figure 3: Performance measurements for IFCC and FSan.

of security, with VTV protecting 95.2% of all possible indirect jumps in our test, and IFCC protecting 99.8%. We have also measured the performance of these approaches and shown that while there is some degradation, averaging in the range of 1%–4%, and in the worst case getting up to 8.7% for VTV (the most precise approach), this penalty is fairly low and seems well within the range of what is acceptable, particularly in exchange for increased security.

Due to our relaxed, yet realistic, attack model coupled with compiler optimizations, we achieve significant performance gains over other CFI implementations while defending against real attacks.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Info. & System Security*, 13(1):4:1–4:40, Oct. 2009.

[2] T. Bao, J. Burket, and M. Woo. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[3] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of ISSTA 2012*, July 2012.

[4] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of SOSP 2009*, Oct. 2009.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of CCS 2010*, pages 559–572. ACM Press, Oct. 2010. URL `https://cs.jhu.edu/~s/papers/noret_ccs2010.html`.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of USENIX Security 1998*, Jan. 1998.

[8] "d0c_s4vage". Insecticides don't kill bugs, Patch Tuesdays do. Online: `http://d0cs4vage.blogspot.com/2011/06/insecticides-dont-kill-bugs-patch.html`, June 2013.

[9] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of NDSS 2012*, Feb. 2012.

[10] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[11] A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Apr. 2001.

[12] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address

spaces. In *Proceedings of OSDI 2006*, pages 75–88, Nov. 2006.

[13] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In P. Stavroulakis and M. Stamp, editors, *Handbook of Information and Communication Security*, pages 633–658. Springer Berlin Heidelberg, 2010.

[14] C. Evans. Exploiting 64-bit linux like a boss. Online: `http://scarybeastsecurity.blogspot.com/search?q=Exploiting+64-bit+linux`, 2013.

[15] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, May 2014.

[16] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[17] Google Developers. Native client. Online: `https://developers.google.com/native-client/`, 2013.

[18] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[19] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of NDSS 2014*. Internet Society, Feb. 2014. To appear.

[20] K. Kortchinsky. 10 years later, which vulnerabilities still matter? Online: `http://ensiwiki.ensimag.fr/images/e/e8/GreHack-2012-talk-Kostya_Kortchinsky_Crypt0ad_-10_years_later_which_in_memory_vulnerabilities_still_matter.pdf`, 2012.

[21] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: better, faster, stronger SFI for the x86. In *Proceedings of PLDI 2012*, pages 395–404, June 2012.

[22] Mozilla Foundation. Mozilla Foundation security advisory 2013-29. Online: `https://www.mozilla.org/security/announce/2013/mfsa2013-29.html`, 2013.

[23] MWR InfoSecurity. Pwn2Own at CanSecWest 2013. Online: `https://labs.mwrinfosecurity.com/blog/2013/03/06/pwn2own-at-cansecwest-2013`, 2013.

[24] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of ASPLOS 2009*, Mar. 2009.

[25] NIST. CVE-2010-0249. Online: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0249`, 2010.

[26] NIST. CVE-2010-3971. Online: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3971`, 2010.

[27] NIST. CVE-2011-1255. Online: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1255`, 2011.

[28] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of CCS 2013*, Nov. 2013.

[29] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of ACSAC 2013*, Dec. 2013.

[30] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security*, 15(1), Mar. 2012.

[31] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of IEEE Symposium on Security and Privacy ("Oakland") 2011*, May 2011.

[32] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of CCS 2011*, Oct. 2011.

[33] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *Proceedings of USENIX Security 2013*, Aug. 2013.

[34] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, L. Szekeres, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, May 2013.

[35] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of USENIX Security 2013*, Aug. 2013.

ret2dir: Rethinking Kernel Isolation

Vasileios P. Kemerlis Michalis Polychronakis Angelos D. Keromytis
Columbia University
{vpk, mikepo, angelos}@cs.columbia.edu

Abstract

Return-to-user (ret2usr) attacks redirect corrupted kernel pointers to data residing in user space. In response, several kernel-hardening approaches have been proposed to enforce a more strict address space separation, by preventing arbitrary control flow transfers and dereferences from kernel to user space. Intel and ARM also recently introduced hardware support for this purpose in the form of the SMEP, SMAP, and PXN processor features. Unfortunately, although mechanisms like the above prevent the explicit sharing of the virtual address space among user processes and the kernel, conditions of implicit sharing still exist due to fundamental design choices that trade stronger isolation for performance.

In this work, we demonstrate how implicit page frame sharing can be leveraged for the complete circumvention of software and hardware kernel isolation protections. We introduce a new kernel exploitation technique, called *return-to-direct-mapped memory (ret2dir)*, which bypasses *all* existing ret2usr defenses, namely SMEP, SMAP, PXN, KERNEXEC, UDEREF, and kGuard. We also discuss techniques for constructing reliable ret2dir exploits against x86, x86-64, AArch32, and AArch64 Linux targets. Finally, to defend against ret2dir attacks, we present the design and implementation of an exclusive page frame ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory pages with minimal runtime overhead.

1 Introduction

Although the operating system (OS) kernel has always been an appealing target, until recently attackers focused mostly on the exploitation of vulnerabilities in server and client applications—which often run with administrative privileges—as they are (for the most part) less complex to analyze and easier to compromise. During the past few years, however, the kernel has become an equally attractive target. Continuing the increasing trend of the previous years, in 2013 there were 355 reported kernel vulnerabilities according to the National Vulnerability Database, 140 more than in 2012 [73]. Admittedly, the exploitation of user-level software has become much harder, as recent versions of popular OSes come with numerous protections and exploit mitigations. The principle of least privilege is better enforced in user accounts and system services, compilers offer more protections against common software flaws, and highly targeted applications, such as browsers and document viewers, have started to employ sandboxing. On the other hand, the kernel has a huge codebase and an attack surface that keeps increasing due to the constant addition of new features [63]. Indicatively, the size of the Linux kernel in terms of lines of code has more than doubled, from 6.6 MLOC in v2.6.11 to 16.9 MLOC in v3.10 [32].

Naturally, instead of putting significant effort to exploit applications fortified with numerous protections and sandboxes, attackers often turn their attention to the kernel. By compromising the kernel, they can elevate privileges, bypass access control and policy enforcement, and escape isolation and confinement mechanisms. For instance, in recent exploits against Chrome and Adobe Reader, after successfully gaining code execution, the attackers exploited kernel vulnerabilities to break out of the respective sandboxed processes [5, 74].

Opportunities for kernel exploitation are abundant. As an example consider the Linux kernel, which has been plagued by common software flaws, such as stack and heap buffer overflows [14, 23, 26], NULL pointer and pointer arithmetic errors [10, 12], memory disclosure vulnerabilities [13, 19], use-after-free and format string bugs [25, 27], signedness errors [17, 24], integer overflows [10,16], race conditions [11,15], as well as missing authorization checks and poor argument sanitization vulnerabilities [18, 20–22]. The exploitation of these bugs is particularly effective, despite the existence of kernel protection mechanisms, due to the weak separation be-

tween user and kernel space. Although user programs cannot access directly kernel code or data, the opposite is not true, as the kernel is mapped into the address space of each process for performance reasons. This design allows an attacker with non-root access to execute code in privileged mode, or tamper-with critical kernel data structures, by exploiting a kernel vulnerability and redirecting the control or data flow of the kernel to code or data in user space. Attacks of this kind, known as *return-to-user (ret2usr)* [57], affect all major OSes, including Windows and Linux, and are applicable in x86/x86-64, ARM, and other popular architectures.

In response to ret2usr attacks, several protections have been developed to enforce strict address space separation, such as PaX's KERNEXEC and UDEREF [77] and kGuard [57]. Having realized the importance of the problem, Intel introduced Supervisor Mode Execute Protection (SMEP) [46] and Supervisor Mode Access Prevention (SMAP) [54], two processor features that, when enabled, prevent the execution (or access) of *arbitrary* user code (or data) by the kernel. ARM has also introduced Privileged Execute-Never (PXN) [4], a feature equivalent to SMEP. These features offer similar guarantees to software protections with negligible runtime overhead.

Although the above mechanisms prevent the explicit sharing of the virtual address space among user processes and the kernel, conditions of *implicit* data sharing still exist. Fundamental OS components, such as physical memory mappings, I/O buffers, and the page cache, can still allow user processes to influence what data is accessible by the kernel. In this paper, we study the above problem in Linux, and expose design decisions that trade stronger isolation for performance. Specifically, we present a new kernel exploitation technique, called *return-to-direct-mapped memory (ret2dir)*, which relies on inherent properties of the memory management subsystem to bypass existing ret2usr protections. This is achieved by leveraging a kernel region that directly maps part or all of a system's physical memory, enabling attackers to essentially "mirror" user-space data within the kernel address space.

The task of mounting a ret2dir attack is complicated due to the different kernel layouts and memory management characteristics of different architectures, the partial mapping of physical memory in 32-bit systems, and the unknown location of the "mirrored" user-space data within the kernel. We present in detail different techniques for overcoming each of these challenges and constructing reliable ret2dir exploits against hardened x86, x86-64, AArch32, and AArch64 Linux targets.

To mitigate the effects of ret2dir attacks, we present the design and implementation of an exclusive page frame ownership scheme for the Linux kernel, which prevents the implicit sharing of physical memory among user processes and the kernel. The results of our evaluation show that the proposed defense offers effective protection with minimal ($<3\%$) runtime overhead.

The main contributions of this paper are the following:

1. We expose a fundamental design weakness in the memory management subsystem of Linux by introducing the concept of *ret2dir* attacks. Our exploitation technique bypasses all existing ret2usr protections (SMEP, SMAP, PXN, KERNEXEC, UDEREF, kGuard) by taking advantage of the kernel's direct-mapped physical memory region.

2. We introduce a detailed methodology for mounting reliable ret2dir attacks against x86, x86-64, AArch32, and AArch64 Linux systems, along with two techniques for forcing user-space exploit payloads to "emerge" within the kernel's direct-mapped RAM area and accurately pinpointing their location.

3. We experimentally evaluate the effectiveness of ret2dir attacks using a set of nine (eight real-world and one artificial) exploits against different Linux kernel configurations and protection mechanisms. In all cases, our transformed exploits bypass successfully the deployed ret2usr protections.

4. We present the design, implementation, and evaluation of an exclusive page frame ownership scheme for the Linux kernel, which mitigates ret2dir attacks with negligible (in most cases) runtime overhead.

2 Background and Related Work

2.1 Virtual Memory Organization in Linux

Designs for safely combining different protection domains range from putting the kernel and user processes into a single address space and establishing boundaries using software isolation [52], to confining user process and kernel components into separate, hardware-enforced address spaces [2, 50, 66]. Linux and Linux-based OSes (Android [47], Firefox OS [72], Chrome OS [48]) adopt a more coarse-grained variant of the latter approach, by dividing the virtual address space into *kernel* and *user* space. In the x86 and 32-bit ARM (AArch32) architectures, the Linux kernel is typically mapped to the upper 1GB of the virtual address space, a split also known as "3G/1G" [28].[1] In x86-64 and 64-bit ARM (AArch64) the kernel is located in the upper *canonical half* [60, 69].

This design minimizes the overhead of crossing protection domains, and facilitates fast user-kernel interactions. When servicing a system call or handling an ex-

[1] Linux also supports 2G/2G and 1G/3G splits. A patch for a 4G/4G split in x86 [53] exists, but was never included in the mainline kernel for performance reasons, as it requires a TLB flush per system call.

ception, the kernel is running within the *context* of a preempted process. Hence, flushing the TLB is not necessary [53], while the kernel can access user space *directly* to read user data or write the result of a system call.

2.2 Return-to-user (ret2usr) Exploits

Although kernel code and user software have both been plagued by common types of vulnerabilities [9], the execution model imposed by the shared virtual memory layout between the kernel and user processes makes kernel exploitation noticeably different. The shared address space provides a unique vantage point to local attackers, as it allows them to control—both in terms of permissions and contents—part of the address space that is accessible by the kernel [91]. Simply put, attackers can easily execute shellcode with kernel rights by hijacking a privileged execution path and redirecting it to user space.

Attacks of this kind, known as return-to-user (ret2usr), have been the de facto kernel exploitation technique (also in non-Linux OSes [88]) for more than a decade [36]. In a ret2usr attack, kernel data is overwritten with user space addresses, typically after the exploitation of memory corruption bugs in kernel code [81], as illustrated in Figure 1. Attackers primarily aim for control data, such as return addresses [86], dispatch tables [36, 44], and function pointers [40, 42, 43, 45], as they directly facilitate arbitrary code execution [89]. Pointers to critical data structures stored in the kernel's heap [38] or the global data section [44] are also common targets, as they allow attackers to tamper with critical data contained in these structures by mapping fake copies in user space [38, 39, 41]. Note that the targeted data structures typically contain function pointers or data that affect the control flow of the kernel, so as to diverge execution to arbitrary points. The end effect of all ret2usr attacks is that *the control or data flow of the kernel is hijacked and redirected to user space code or data* [57].

Most ret2usr exploits use a multi-stage shellcode, with a first stage that lies in user space and "glues" together kernel functions (i.e., the second stage) for performing privilege escalation or executing a rootshell. Technically, ret2usr expects the kernel to run within the context of a process controlled by the attacker for exploitation to be reliable. However, kernel bugs have also been identified and exploited in interrupt service routines [71]. In such cases, where the kernel is either running in *interrupt context* or in a process context beyond the attacker's control [37, 85], the respective shellcode has to be injected in kernel space or be constructed using code gadgets from the kernel's text in a ROP/JOP fashion [8, 51, 87]. The latter approach is gaining popularity in real-world exploits, due to the increased adoption of kernel hardening techniques [31, 65, 68, 92, 93, 95].

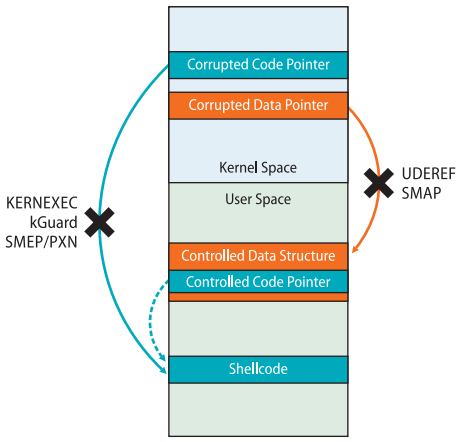


Figure 1: Operation of ret2usr attacks. A kernel code or data pointer is hijacked and redirected to controlled code or data in user space (tampered-with data structures may further contain pointers to code). Various protection mechanisms (KERNEXEC, UDEREF, kGuard, SMEP, SMAP, PXN) prevent arbitrary control flow transfers and dereferences from kernel to user space.

2.3 Protections Against ret2usr Attacks

Return-to-user attacks are yet another incarnation of the confused deputy problem [49]. Given the multi-architecture [42, 83] and multi-OS [88] nature of the problem, several defensive mechanisms exist for it. In the remainder of this section, we discuss the ret2usr defenses available in Linux with the help of Figure 1.

PaX: KERNEXEC and UDEREF are two features of the PaX [77] hardening patch set that prevent control flow transfers and dereferences from kernel to user space. In x86, KERNEXEC and UDEREF rely on memory segmentation [78] to map the kernel space into a 1GB segment that returns a memory fault whenever privileged code tries to dereference pointers to, or fetch instructions from, non-kernel addresses. In x86-64, due to the lack of segmentation support, UDEREF/amd64 [79] remaps user space memory into a different (shadow), non-executable area when execution enters the kernel (and restores it on exit), to prevent user-space dereferences. As the overhead of remapping memory is significant, an alternative for x86-64 systems is to enable KERNEXEC/amd64 [80], which has much lower overhead, but offers protection against only control-flow hijacking attacks. Recently, KERNEXEC and UDEREF were ported to the ARM architecture [90], but the patches added support for AArch32 only and rely on the deprecated MMU domains feature (discussed below).

SMEP/SMAP/PXN: Supervisor Mode Execute Protection (SMEP) [46] and Supervisor Mode Access Prevention (SMAP) [54] are two recent features of Intel

processors that facilitate stronger address space separation (latest kernels support both features [31,95]). SMEP provides analogous protection to KERNEXEC, whereas SMAP operates similarly to UDEREF. Recently, ARM added support for an SMEP-equivalent feature, dubbed Privileged Execute-Never (PXN) [4], but Linux uses it only on AArch64. More importantly, on AArch32, PXN requires the MMU to operate on LPAE mode (the equivalent of Intel's Physical Address Extension (PAE) mode [55]), which disables MMU domains. Therefore, the use of KERNEXEC/UDEREF on AArch32 implies giving up support for PXN and large memory ($>$ 4GB).

kGuard: kGuard [57] is a cross-platform compiler extension that protects the kernel from ret2usr attacks without relying on special hardware features. It enforces lightweight address space segregation by augmenting (at compile time) potentially exploitable control transfers with dynamic control-flow assertions (CFAs) that (at runtime) prevent the unconstrained transition of privileged execution paths to user space. The injected CFAs perform a small runtime check before every computed branch to verify that the target address is always located in kernel space or loaded from kernel-mapped memory. In addition, kGuard incorporates code diversification techniques for thwarting attacks against itself.

3 Attack Overview

Linux follows a design that trades weaker kernel-to-user segregation in favor of faster interactions between user processes and the kernel. The ret2usr protections discussed in the previous section aim to alleviate this design weakness, and fortify the isolation between kernel and user space with minimal overhead. In this work, we seek to assess the security offered by these protections and investigate whether certain performance-oriented design choices can render them ineffective. Our findings indicate that there exist fundamental decisions, deeply rooted into the architecture of the Linux memory management subsystem (`mm`), which can be abused to weaken the isolation between kernel and user space. We introduce a novel kernel exploitation technique, named return-to-direct-mapped memory (ret2dir), which allows an attacker to perform the equivalent of a ret2usr attack on a hardened system.

3.1 Threat Model

We assume a Linux kernel hardened against ret2usr attacks using one (or a combination) of the protection mechanisms discussed in Section 2.3. Moreover, we assume an *unprivileged* attacker with local access, who seeks to elevate privileges by exploiting a kernel-memory corruption vulnerability [10–27] (see Section 2.2). Note that we do not make any assumptions about the type of corrupted data—code and data pointers are both possible targets [36,40,42–45,86]. Overall, the adversarial capabilities we presume are identical to those needed for carrying out a ret2usr attack.

3.2 Attack Strategy

In a kernel hardened against ret2usr attacks, the hijacked control or data flow can no longer be redirected to user space in a direct manner—the respective ret2usr protection scheme(s) will block any such attempt, as shown in Figure 1. However, the implicit physical memory sharing between user processes and the kernel allows an attacker to *deconstruct* the isolation guarantees offered by ret2usr protection mechanisms, and redirect the kernel's control or data flow to user-controlled code or data.

A key facility that enables the implicit sharing of physical memory is *`physmap`*: a large, contiguous virtual memory region inside kernel address space that contains a direct mapping of part or all (depending on the architecture) physical memory. This region plays a crucial role in enabling the kernel to allocate and manage dynamic memory as fast as possible (we discuss the structure of `physmap` in Section 4). We should stress that although in this study we focus on Linux—one of the most widely used OSes—direct-mapped RAM regions exist (in some form) in many OSes, as they are considered standard practice in physical memory management. For instance, Solaris uses the `seg_kpm` mapping facility to provide a direct mapping of the whole RAM in 64-bit architectures [70].

As physical memory is allotted to user processes and the kernel, the existence of `physmap` results in *address aliasing*. Virtual address aliases, or *synonyms* [62], occur when two or more different virtual addresses map to the same physical memory address. Given that `physmap` maps a large part (or all) of physical memory within the kernel, the memory of an attacker-controlled user process is accessible through its kernel-resident synonym.

The first step in mounting a ret2dir attack is to map in user space the exploit *payload*. Depending on whether the exploited vulnerability enables the corruption of a code pointer [36,40,42–45,86] or a data pointer [38,39,41], the payload will consist of either shellcode, or controlled (tampered-with) data structures, as shown in Figure 2. Whenever the `mm` subsystem allocates (dynamic) memory to user space, it actually defers giving page frames until the very last moment. Specifically, physical memory is granted to user processes in a lazy manner, using the *demand paging* and *copy-on-write* methods [7], which both rely on page faults to actually allocate RAM. When the content of the payload is initialized, the MMU generates a page fault, and the kernel allocates a page

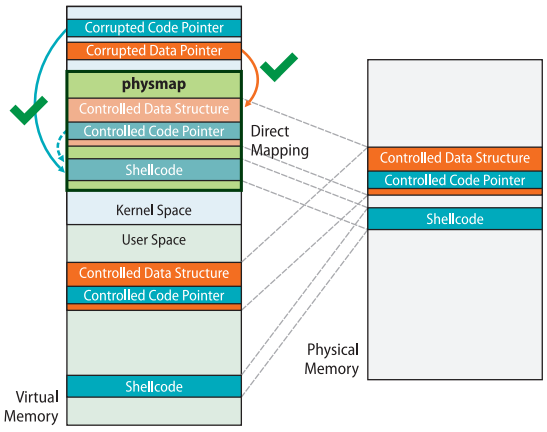


Figure 2: Overall ret2dir operation. The `physmap` (direct-mapped RAM) area enables a hijacked kernel code or data pointer to access user-controlled data, without crossing the user-kernel space boundary.

frame to the attacking process. Page frames are managed by `mm` using a *buddy allocator* [61]. Given the existence of `physmap`, the moment the buddy allocator provides a page frame to be mapped in user space, `mm` effectively creates an alias of the exploit payload in kernel space, as shown in Figure 2. Although the kernel never uses such synonyms directly, `mm` keeps the whole RAM premapped in order to boost page frame reclamation. This allows newly deallocated page frames to be made available to the kernel instantly, without the need to alter page tables (see Section 4.1 for more details).

Overall, ret2dir takes advantage of the implicit data sharing between user and kernel space (due to `physmap`) to redirect a hijacked kernel control or data flow to a set of kernel-resident synonym pages, effectively performing the equivalent of a ret2usr attack without reaching out to user space. It is important to note that the malicious payload "emerges" in kernel space the moment a page frame is given to the attacking process. The attacker does not have to explicitly "push" (copy) the payload to kernel space (e.g., via pipes or message queues), as `physmap` makes it readily available. The use of such methods is also much less flexible, as the system imposes strict limits to the amount of memory that can be allocated for kernel-resident buffers, while the exploit payload will (most likely) have to be encapsulated in certain kernel data objects that can affect its structure.

4 Demystifying `physmap`

A critical first step in understanding the mechanics of ret2dir attacks is to take a look at how the address space of the Linux kernel is organized—we use the x86 platform as a reference. The x86-64 architecture uses 48-bit virtual addresses that are sign-extended to 64 bits (i.e., bits `[48:63]` are copies of bit `[47]`). This scheme natively splits the 64-bit virtual address space in two canonical halves of 128TB each. Kernel space occupies the upper half (`0xFFFF800000000000` – `0xFFFFFFFFFFFFFFFF`), and is further divided into six regions [60]: the `fixmap` area, modules, kernel image, `vmemmap` space, `vmalloc` arena, and `physmap`. In x86, on the other hand, the kernel space can be assigned to the upper 1GB, 2GB, or 3GB part of the address space, with the first option being the default. As kernel virtual address space is limited, it can become a scarce resource, and certain regions collide to prevent its waste (e.g., modules and `vmalloc` arena, kernel image and `physmap`).[2] For the purposes of ret2dir, in the following, we focus only on the direct-mapped region.

4.1 Functionality

The `physmap` area is a mapping of paramount importance to the performance of the kernel, as it facilitates dynamic kernel memory allocation. At a high level, `mm` offers two main methods for requesting memory: `vmalloc` and `kmalloc`. With the `vmalloc` family of routines, memory can only be allocated in multiples of page size and is guaranteed to be virtually contiguous but *not* physically contiguous. In contrast, with the `kmalloc` family of routines, memory can be allocated in byte-level chunks, and is guaranteed to be *both* virtually and physically contiguous.

As it offers memory only in page multiples, `vmalloc` leads to higher internal memory fragmentation and often poor cache performance. More importantly, `vmalloc` needs to alter the kernel's page tables every time memory is (de)allocated to map or unmap the respective page frames to or from the `vmalloc` arena. This not only incurs additional overhead, but results in increased TLB thrashing [67]. For these reasons, the majority of kernel components use `kmalloc`. However, given that `kmalloc` can be invoked from any context, including that of interrupt service routines, which have strict timing constraints, it must satisfy a multitude of different (and contradicting) requirements. In certain contexts, the allocator should never sleep (e.g., when locks are held). In other cases, it should never fail, or it should return memory that is guaranteed to be physically contiguous (e.g., when a device driver reserves memory for DMA).

Given constraints like the above, *`physmap` is a necessity, as the main facilitator of optimal performance.* The `mm` developers opted for a design that lays `kmalloc`

[2] To access the contents of a page frame, the kernel must first map that frame in kernel space. In x86, however, the kernel has only 1GB – 3GB virtual addresses available for managing (up to) 64GB of RAM.

Architecture		PHYS_OFFSET	Size	Prot.
x86	(3G/1G)	0xC0000000	891MB	RW
	(2G/2G)	0x80000000	1915MB	RW
	(1G/3G)	0x40000000	2939MB	RW
AArch32	(3G/1G)	0xC0000000	760MB	RW**X**
	(2G/2G)	0x80000000	1784MB	RW**X**
	(1G/3G)	0x40000000	2808MB	RW**X**
x86-64		0xFFFF880000000000	64TB	RW(**X**)
AArch64		0xFFFFFFC000000000	256GB	RW**X**

Table 1: `physmap` characteristics across different architectures (x86, x86-64, AArch32, AArch64).

over a region[3] that pre-maps the entire RAM (or part of it) for the following reasons [7]. First, `kmalloc` (de)allocates memory without touching the kernel's page table. This not only reduces TLB pressure significantly, but also removes high-latency operations, like page table manipulation and TLB shootdowns [70], from the fast path. Second, the linear mapping of page frames results in virtual memory that is guaranteed, by design, to be always physically contiguous. This leads to increased cache performance, and has the added benefit of allowing drivers to directly assign `kmalloc`'ed regions to DMA devices that can only operate on physically contiguous memory (e.g., when there is no IOMMU support). Finally, page frame accounting is greatly simplified, as address translations (virtual-to-physical and vice versa) can be performed using solely arithmetic operations [64].

4.2 Location and Size

The `physmap` region is an architecture-independent feature (this should come as no surprise given the reasons we outlined above) that exists in all popular Linux platforms. Depending on the memory addressing characteristics of each ISA, the size of `physmap` and its exact location may differ. Nonetheless, in all cases: *(i)* there exists a *direct* mapping of part or all physical memory in kernel space, and *(ii)* the mapping starts at a *fixed*, known location. The latter is true even in the case where kernel-space ASLR (KASLR) [35] is employed.

Table 1 lists `physmap`'s properties of interest for the platforms we consider. In x86-64 systems, the `physmap` maps directly in a 1:1 manner, starting from page frame zero, the entire RAM of the system into a 64TB region. AArch64 systems use a 256GB region for the same purpose [69]. Conversely, in x86 systems, the kernel directly maps only a portion of the available RAM.

The size of `physmap` on 32-bit architectures depends on two factors: *(i)* the user/kernel split used (3G/1G, 2G/2G, or 1G/3G), and *(ii)* the size of the `vmalloc` arena. Under the default setting, in which 1GB is assigned to kernel space and the `vmalloc` arena occupies 120MB, the size of `physmap` is 891MB (1GB - `sizeof(vmalloc + pkmap + fixmap + unused)`) and starts at `0xC0000000`. Likewise, under a 2G/2G (1G/3G) split, `physmap` starts at `0x80000000` (`0x40000000`) and spawns 1915MB (2939MB). The situation in AArch32 is quite similar [59], with the only difference being the default size of the `vmalloc` arena (240MB).

Overall, in 32-bit systems, the amount of directly mapped physical memory depends on the size of RAM and `physmap`. If `sizeof(physmap)` $\geq$ `sizeof(RAM)`, then the entire RAM is direct-mapped—a common case for 32-bit mobile devices with up to 1GB of RAM. Otherwise, only up to `sizeof(physmap)/sizeof(PAGE)` pages are mapped directly, starting from the first page frame.

4.3 Access Rights

A crucial aspect for mounting a ret2dir attack is the memory access rights of `physmap`. To get the protection bits of the kernel pages that correspond to the direct-mapped memory region, we built `kptdump`:[4] a utility in the form of a kernel module that exports page tables through the `debugfs` pseudo-filesystem [29]. The tool traverses the kernel page table, available via the global symbols `swapper_pg_dir` (x86/AArch32/AArch64) and `init_level4_pgt` (x86-64), and dumps the flags (`U/S`, `R/W`, `XD`) of every kernel page that falls within the `physmap` region.

In x86, `physmap` is mapped as "readable and writeable" (`RW`) in all kernel versions we tried (the oldest one was v2.6.32, released on Dec. 2009). In x86-64, however, the permissions of `physmap` are *not* in sane state. Kernels up to v3.8.13 violate the `W^X` property by mapping the entire region as "readable, writeable, and executable" (`RWX`)—only very recent kernels ($\geq$ v3.9) use the more conservative `RW` mapping. Finally, AArch32 and AArch64 map `physmap` with `RWX` permissions in every kernel version we tested (up to v3.12).

[3] `kmalloc` is not directly layered over `physmap`. It is instead implemented as a collection of geometrically distributed (32B–4KB) *slabs*, which are in turn placed over `physmap`. The slab layer is a hierarchical, type-based data structure caching scheme. By taking into account certain factors, such as page and object sizes, cache line information, and memory access times (in NUMA systems), it can perform intelligent allocation choices that minimize memory fragmentation and make the best use of a system's cache hierarchy. Linux adopted the slab allocator of SunOS [6], and as of kernel v3.12, it supports three variants: `SLAB`, `SLUB` (default), and `SLOB`.

[4] `kptdump` resembles the functionality of Arjan van de Ven's patch [94]; unfortunately, we had to resort to a custom solution, as that patch is only available for x86/x86-64 and cannot be used "as-is" in any other architecture.

5 Locating Synonyms

The final piece for mounting a ret2dir exploit is finding a way to reliably pinpoint the location of a synonym address in the physmap area, given its user-space counterpart. For legacy environments, in which physmap maps only part of the system's physical memory, such as a 32-bit system with 8GB of RAM, an additional requirement is to ensure that the synonym of a user-space address of interest exists. We have developed two techniques for achieving both goals. The first relies on page frame information available through the pagemap interface of the /proc filesystem, which is currently accessible by non-privileged users in all Linux distributions that we studied. As the danger of ret2dir attacks will (hopefully) encourage system administrators and Linux distributions to disable access to pagemap, we have developed a second technique that does not rely on any information leakage from the kernel.

5.1 Leaking PFNs (via /proc)

The procfs pseudo-filesystem [58] has a long history of leaking security-sensitive information [56, 76]. Starting with kernel v2.6.25 (Apr. 2008), a set of pseudo-files, including /proc/<pid>/pagemap, were added in /proc to enable the examination of page tables for debugging purposes. To assess the prevalence of this facility, we tested the latest releases of the most popular distributions according to DistroWatch [34] (i.e., Debian, Ubuntu, Fedora, and CentOS). In all cases, pagemap was enabled by default.

For every user-space page, pagemap provides a 64-bit value, indexed by (virtual) page number, which contains information regarding the presence of the page in RAM [1]. If a page is present in RAM, then bit [63] is set and bits [0:54] encode its page frame number (PFN). That being so, the PFN of a given user-space virtual address uaddr, can be located by opening /proc/<pid>/pagemap and reading eight bytes from file offset (uaddr/4096) * sizeof(uint64_t) (assuming 4KB pages).

Armed with the PFN of a given uaddr, denoted as PFN(uaddr), its synonym SYN(uaddr) in physmap can be located using the following formula: SYN(uaddr) = PHYS_OFFSET + 4096 * (PFN(uaddr) − PFN_MIN). PHYS_OFFSET corresponds to the known, fixed starting kernel virtual address of physmap (values for different configurations are shown in Table 1), and PFN_MIN is the first page frame number—in many architectures, including ARM, physical memory starts from a non-zero offset (e.g., 0x60000000 in Versatile Express ARM boards, which corresponds to PFN_MIN = 0x60000). To prevent SYN(uaddr) from being reclaimed (e.g., after swapping out uaddr), the respective user page can be "pinned" to RAM using mlock.

sizeof(RAM) > sizeof(physmap): For systems in which part of RAM is direct-mapped, only a subset of PFNs is accessible through physmap. For instance, in an x86 system with 4GB of RAM, the PFN range is 0x0–0x100000. However, under the default 3G/1G split, the physmap region maps only the first 891MB of RAM (see Table 1 for other setups), which means PFNs from 0x0 up to 0x37B00 (PFN_MAX). If the PFN of a user-space address is greater than PFN_MAX (the PFN of the last direct-mapped page), then physmap does not contain a synonym for that address. Naturally, the question that arises is whether we can *force* the buddy allocator to provide page frames with PFNs less than PFN_MAX.

For compatibility reasons, mm splits physical memory into several *zones*. In particular, DMA processors of older ISA buses can only address the first 16MB of RAM, while some PCI DMA peripherals can access only the first 4GB. To cope with such limitations, mm supports the following zones: ZONE_DMA, ZONE_DMA32, ZONE_NORMAL, and ZONE_HIGHMEM. The latter is available in 32-bit platforms and contains the page frames that cannot be directly addressed by the kernel (i.e., those that are not mapped in physmap). ZONE_NORMAL contains page frames above ZONE_DMA (and ZONE_DMA32, in 64-bit systems) and below ZONE_HIGHMEM. When only part of RAM is direct-mapped, mm orders the zones as follows: ZONE_HIGHMEM > ZONE_NORMAL > ZONE_DMA. Given a page frame request, mm will try to satisfy it starting with the highest zone that complies with the request (e.g., as we have discussed, the direct-mapped memory of ZONE_NORMAL is preferred for kmalloc), moving towards lower zones as long as there are no free page frames available.

From the perspective of an attacker, user processes get their page frames from ZONE_HIGHMEM first, as mm tries to preserve the page frames that are direct-mapped for dynamic memory requests from the kernel. However, when the page frames of ZONE_HIGHMEM are depleted, due to increased memory pressure, *mm inevitably starts providing page frames from ZONE_NORMAL or ZONE_DMA*. Based on this, our strategy is as follows. The attacking process repeatedly uses mmap to request memory. For each page in the requested memory region, the process causes a page fault by accessing a single byte, forcing mm to allocate a page frame (alternatively, the MAP_POPULATE flag in mmap can be used to pre-allocate all the respective page frames). The process then checks the PFN of every allocated page, and the same procedure is repeated until a PFN less than PFN_MAX is

obtained. The synonym of such a page is then guaranteed to be present in physmap, and its exact address can be calculated using the formula presented above. Note that depending on the size of physical memory and the user/kernel split used, we may have to spawn additional processes to completely deplete ZONE_HIGHMEM. For example, on an x86 machine with 8GB of RAM and the default 3G/1G split, up to three processes might be necessary to guarantee that a page frame that falls within physmap will be acquired. Interestingly, the more benign processes are running on the system, the easier it is for an attacker to acquire a page with a synonym in physmap; additional tasks create memory pressure, "pushing" the attacker's allocations to the desired zones.

Contiguous synonyms: Certain exploits may require more than a single page for their payload(s). Pages that are virtually contiguous in user space, however, do not necessarily map to page frames that are physically contiguous, which means that their synonyms will not be contiguous either. Yet, given physmap's linear mapping, two pages with consecutive synonyms have PFNs that are sequential. Therefore, if 0xBEEF000 and 0xFEEB000 have PFNs 0x2E7C2 and 0x2E7C3, respectively, then they are contiguous in physmap despite being ~64MB apart in user space.

To identify consecutive synonyms, we proceed as follows. Using the same methodology as above, we compute the synonym of a random user page. We then repeatedly obtain more synonyms, each time comparing the PFN of the newly acquired synonym with the PFNs of those previously retrieved. The process continues until any two (or more, depending on the exploit) synonyms have sequential PFNs. The exploit payload can then be split appropriately across the user pages that correspond to synonyms with sequential PFNs.

5.2 physmap Spraying

As eliminating access to /proc/<pid>/pagemap is a rather simple task, we also consider the case in which PFN information is not available. In such a setting, given a user page that is present in RAM, there is no direct way of determining the location of its synonym inside physmap. Recall that our goal is to identify a kernel-resident page in the physmap area that "mirrors" a user-resident exploit payload. Although we cannot identify the synonym of a given user address, it is still possible to proceed in the opposite direction: pick an *arbitrary* physmap address, and ensure (to the extent possible) that its corresponding page frame is mapped by a user page that contains the exploit payload.

This can be achieved by exhausting the address space of the attacking process with (aligned) copies of the exploit payload, in a way similar to heap spraying [33]. The base address and length of the physmap area is known in advance (Table 1). The latter corresponds to PFN_MAX − PFN_MIN page frames, shared among all user processes and the kernel. If the attacking process manages to copy the exploit payload into N memory-resident pages (in the physical memory range mapped by physmap), then the probability (P) that an arbitrarily chosen, page-aligned physmap address will point to the exploit payload is: P = N / (PFN_MAX − PFN_MIN). Our goal is to maximize P.

Spraying: Maximizing N is straightforward, and boils down to acquiring as many page frames as possible. The technique we use is similar to the one presented in Section 5.1. The attacking process repeatedly acquires memory using mmap and "sprays" the exploit payload into the returned regions. We prefer using mmap, over ways that involve shmget, brk, and remap_file_pages, due to system limits typically imposed on the latter. MAP_ANONYMOUS allocations are also preferred, as existing file-backed mappings (from competing processes) will be swapped out with higher priority compared to anonymous mappings. The copying of the payload causes page faults that result in page frame allocations by mm (alternatively MAP_POPULATE can be used). If the virtual address space is not enough for depleting the entire RAM, as is the case with certain 32-bit configurations, the attacking process must spawn additional child processes to assist with the allocations.

The procedure continues until mm starts swapping "sprayed" pages to disk. To pinpoint the exact moment that swapping occurs, each attacking process checks periodically whether its sprayed pages are still resident in physical memory, by calling the getrusage system call every few mmap invocations. At the same time, all attacking processes start a set of background threads that repeatedly write-access the already allocated pages, simulating the behavior of mlock, and preventing (to the extent possible) sprayed pages from being swapped out—mm swaps page frames to disk using the LRU policy. Hence, by accessing pages repeatedly, mm is tricked to believe that they correspond to fresh content. When the number of memory-resident pages begins to drop (i.e., the resident-set size (RSS) of the attacking process(es) starts decreasing), the maximum allowable physical memory footprint has been reached. Of course, the size of this footprint also depends on the memory load inflicted by other processes, which compete with the attacking processes for RAM.

Signatures: As far as PFN_MAX − PFN_MIN is concerned, we can reduce the set of potential target pages in the physmap region, by excluding certain pages that correspond to frames that the buddy allocator will never provide to user space. For example, in x86 and x86-64, the BIOS typically stores the hardware configuration de-

tected during POST at page frame zero. Likewise, the physical address range 0xA0000–0xFFFFF is reserved for mapping the internal memory of certain graphics cards. In addition, the ELF sections of the kernel image that correspond to kernel code and global data are loaded at known, fixed locations in RAM (e.g., 0x1000000 in x86). Based on these and other predetermined allocations, we have generated *physmap signatures* of reserved page frame ranges for each configuration we consider. If a signature is not available, then all page frames are potential targets. By combining physmap spraying and signatures, we can maximize the probability that our informed selection of an arbitrary page from physmap will point to the exploit payload. The results of our experimental security evaluation (Section 7) show that, depending on the configuration, the probability of success can be as high as 96%.

6 Putting It All Together

6.1 Bypassing SMAP and UDEREF

We begin with an example of a ret2dir attack against an x86 system hardened with SMAP or UDEREF. We assume an exploit for a kernel vulnerability that allows us to corrupt a kernel *data* pointer, named kdptr, and overwrite it with an arbitrary value [38,39,41]. On a system with an unhardened kernel, an attacker can overwrite kdptr with a user-space address, and force the kernel to dereference it by invoking the appropriate interface (e.g., a buggy system call). However, the presence of SMAP or UDEREF will cause a memory access violation, effectively blocking the exploitation attempt. To overcome this, a ret2dir attack can be mounted as follows.

First, an attacker-controlled user process reserves a single page (4KB), say at address 0xBEEF000. Next, the process moves on to initialize the newly allocated memory with the exploit payload (e.g., a tampered-with data structure). This payload initialization phase will cause a page fault, triggering mm to request a free page frame from the buddy allocator and map it at address 0xBEEF000. Suppose that the buddy system picks page frame 1904 (0x770). In x86, under the default 3G/1G split, physmap starts at 0xC0000000, which means that the page frame has been pre-mapped at address 0xC0000000 + (4096 * 0x770) = 0xC0770000 (according to formula in Section 5.1). At this point, 0xBEEF000 and 0xC0770000 are synonyms; they both map to the physical page that contains the attacker's payload. Consequently, any data in the area 0xBEEF000–0xBEEFFFFF is readily accessible by the kernel through the synonym addresses 0xC0770000–0xC0770FFF. To make matters worse, given that physmap is primarily used for implementing dynamic memory, the kernel cannot distinguish whether the kernel data structure located at address 0xC0770000 is fake or legitimate (i.e., properly allocated using kmalloc). Therefore, by overwriting kdptr with 0xC0770000 (instead of 0xBEEF000), the attacker can bypass SMAP and UDEREF, as both protections consider benign any dereference of memory addresses above 0xC0000000.

6.2 Bypassing SMEP, PXN, KERNEXEC, and kGuard

We use a running example from the x86-64 architecture to demonstrate how a ret2dir attack can bypass KERNEXEC, kGuard, and SMEP (PXN operates almost identically to SMEP). We assume the exploitation of a kernel vulnerability that allows the corruption of a kernel function pointer, namely kfptr, with an arbitrary value [40, 42, 43, 45]. In this setting, the exploit payload is not a set of fake data structures, but machine code (shellcode) to be executed with elevated privilege. In real-world kernel exploits, the payload typically consists of a multi-stage shellcode, the first stage of which stitches together kernel routines (second stage) for performing privilege escalation [89]. In most cases, this boils down to executing something similar to commit_creds(prepare_kernel_cred(0)). These two routines replace the credentials ((e)uid, (e)gid) of a user task with zero, effectively granting root privileges to the attacking process.

The procedure is similar as in the previous example. Suppose that the payload has been copied to user-space address 0xBEEF000, which the buddy allocator assigned to page frame 190402 (0x2E7C2). In x86-64, physmap starts at 0xFFFF880000000000 (see Table 1), and maps the whole RAM using regular pages (4KB). Hence, a synonym of address 0xBEEF000 is located within kernel space at address 0xFFFF880000000000 + (4096 * 0x2E7C2) = 0xFFFF87FF9F080000.

In ret2usr scenarios where attackers control a kernel function pointer, an advantage is that they also control the memory access rights of the user page(s) that contain the exploit payload, making it trivially easy to mark the shellcode as executable. In a hardened system, however, a ret2dir attack allows controling only the *content* of the respective synonym pages within physmap—not their permissions. In other words, although the attacker can set the permissions of the range 0xBEEF000 – 0xBEEFFFF, this will *not* affect the access rights of the corresponding physmap pages.

Unfortunately, as shown in Table 1, the W^X property is not enforced in many platforms, including x86-64. In our example, the content of user ad-

dress 0xBEEF000 is also accessible through kernel address 0xFFFF87FF9F080000 as plain, executable code. Therefore, by simply overwriting kfptr with 0xFFFF87FF9F080000 and triggering the kernel to dereference it, an attacker can directly execute shellcode with kernel privileges. KERNEXEC, kGuard, and SMEP (PXN) cannot distinguish whether kfptr points to malicious code or a legitimate kernel routine, and as long as kfptr $\geq$ 0xFFFF880000000000 and *kfptr is RWX, the dereference is considered benign.

Non-executable physmap: In the above example, we took advantage of the fact that some platforms map part (or all) of the physmap region as executable (X). The question that arises is whether ret2dir can be effective when physmap has sane permissions. As we demonstrate in Section 7, even in this case, ret2dir attacks are possible through the use of return-oriented programming (ROP) [8, 51, 87].

Let's revisit the previous example, this time under the assumption that physmap is not executable. Instead of mapping regular shellcode at 0xBEEF000, an attacker can map an equivalent ROP payload: an implementation of the same functionality consisting solely of a chain of code fragments ending with ret instructions, known as gadgets, which are located in the kernel's (executable) text segment. To trigger the ROP chain, kfptr is overwritten with an address that points to a *stack pivoting* gadget, which is needed to set the stack pointer to the beginning of the ROP payload, so that each gadget can transfer control to the next one. By overwriting kfptr with the address of a pivot sequence, like xchg %rax, %rsp; ret (assuming that %rax points to 0xFFFF87FF9F080000), the synonym of the ROP payload now acts as a kernel-mode stack. Note that Linux allocates a separate kernel stack for every user thread using kmalloc, making it impossible to differentiate between a legitimate stack and a ROP payload "pushed" in kernel space using ret2dir, as both reside in physmap. Finally, the ROP code should also take care of restoring the stack pointer (and possibly other CPU registers) to allow for reliable kernel continuation [3, 81].

7 Security Evaluation

7.1 Effectiveness

We evaluated the effectiveness of ret2dir against kernels hardened with ret2usr protections, using real-world and custom exploits. We obtained a set of eight ret2usr exploits from the Exploit Database (EDB) [75], covering a wide range of kernel versions (v2.6.33.6–v3.8). We ran each exploit on an unhardened kernel to verify that it works, and that it indeed follows a ret2usr exploitation approach. Next, we repeated the same experiment with every kernel hardened against ret2usr attacks, and, as expected, all exploits failed. Finally, we transformed the exploits into ret2dir-equivalents, using the technique(s) presented in Section 5, and used them against the same hardened systems. Overall, our ret2dir versions of the exploits *bypassed all available ret2usr protections*, namely SMEP, SMAP, PXN, KERNEXEC, UDEREF, and kGuard.

Table 2 summarizes our findings. The first two columns (EDB-ID and CVE) correspond to the tested exploit, and the third (Arch.) and fourth (Kernel) denote the architecture and kernel version used. The Payload column indicates the type of payload pushed in kernel space using ret2dir, which can be a ROP payload (ROP), executable instructions (SHELLCODE), tampered-with data structures (STRUCT), or a combination of the above, depending on the exploit. The Protection column lists the deployed protection mechanisms in each case. Empty cells correspond to protections that are not applicable in the given setup, because they may not be *(i)* available for a particular architecture, *(ii)* supported by a given kernel version, or *(iii)* relevant against certain types of exploits. For instance, PXN is available only in ARM architectures, while SMEP and SMAP are Intel processor features. Furthermore, support for SMEP was added in kernel v3.2 and for SMAP in v3.7. Note that depending on the permissions of the physmap area (see Table 1), we had to modify some of the exploits that relied on plain shellcode to use a ROP payload, in order to achieve arbitrary code execution (although in ret2usr exploits attackers can give executable permission to the user-space memory that contains the payload, in ret2dir exploits it is not possible to modify the permissions of physmap).[5] Entries for kGuard marked with $\star$ require access to the (randomized) text section of the respective kernel.

As we mentioned in Section 2.3, KERNEXEC and UDEREF were recently ported to the AArch32 architecture [90]. In addition to providing stronger address space separation, the authors made an effort to fix the permissions of the kernel in AArch32, by enforcing the W^X property for the majority of RWX pages in physmap. However, as the respective patch is currently under development, there still exist regions inside physmap that are mapped as RWX. In kernel v3.8.7, we identified a ~6MB physmap region mapped as RWX that enabled the execution of plain shellcode in our ret2dir exploit.

The most recent kernel version for which we found a publicly-available exploit is v3.8. Thus, to evaluate the latest kernel series (v3.12) we used a custom exploit. We

[5] Exploit 15285 uses ROP code to bypass KERNEXEC/UDEREF and plain shellcode to evade kGuard. Exploit 26131 uses ROP code in x86 (kernel v3.5) to bypass KERNEXEC/UDEREF and SMEP/SMAP, and plain shellcode in x86-64 (kernel v3.8) to bypass kGuard, KERNEXEC, and SMEP.

EDB-ID	CVE	Arch.	Kernel	Payload	Protection						Bypassed
26131	2013-2094	x86/x86-64	3.5/3.8	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard	SMEP	SMAP		✓
24746	2013-1763	x86-64	3.5	SHELLCODE	KERNEXEC		kGuard	SMEP			✓
15944	N/A	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
15704	2010-4258	x86	2.6.35.8	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
15285	2010-3904	x86-64	2.6.33.6	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
15150	2010-3437	x86	2.6.35.8	STRUCT		UDEREF					✓
15023	2010-3301	x86-64	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
14814	2010-2959	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*				✓
Custom	N/A	x86	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*	SMEP	SMAP		✓
Custom	N/A	x86-64	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*	SMEP	SMAP		✓
Custom	N/A	AArch32	3.8.7	STRUCT+SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
Custom	N/A	AArch64	3.12	STRUCT+SHELLCODE			kGuard			PXN	✓

Table 2: Tested exploits (converted to use the ret2dir technique) and configurations.

x86-64

```
push   %rbp
mov    %rsp,      %rbp
push   %rbx
mov    $<pkcred>, %rbx
mov    $<ccreds>, %rax
mov    $0x0,      %rdi
callq  *%rax
mov    %rax,      %rdi
callq  *%rbx
mov    $0x0,      %rax
pop    %rbx
leaveq
ret
```

AArch32

```
push  r3, lr
mov   r0, #0
ldr   r1, [pc, #16]
blx   r1
pop   r3, lr
ldr   r1, [pc, #8]
bx    r1
<pkcred>
<ccreds>
```

Figure 3: The plain shellcode used in ret2dir exploits for x86-64 (left) and AArch32 (right) targets (`pkcred` and `ccreds` correspond to the addresses of `prepare_kernel_cred` and `commit_creds`).

```
                /* save orig. esp                */
0xc10ed359      /* pop    %edx          ; ret    */
<SCRATCH_SPACE_ADDR1>
0xc127547f      /* mov    %eax, (%edx) ; ret    */
                /* save orig. ebp        */
0xc10309d5      /* xchg   %eax, %ebp   ; ret    */
0xc10ed359      /* pop    %edx          ; ret    */
<SCRATCH_SPACE_ADDR2>
0xc127547f      /* mov    %eax, (%edx) ; ret    */
        /* commit_creds(prepare_kernel_cred(0) */
0xc1258894      /* pop    %eax          ; ret    */
0x00000000
0xc10735e0      /* addr. of prepare_kernel_cred */
0xc1073340      /* addr. of commit_creds'       */
                /* restore the saved CPU state  */
0xc1258894      /* pop    %eax          ; ret    */
<SCRATCH_SPACE_ADDR2>
0xc1036551      /* mov  (%eax), %eax   ; ret    */
0xc10309d5      /* xchg   %eax, %ebp   ; ret    */
0xc1258894      /* pop    %eax          ; ret    */
<SCRATCH_SPACE_ADDR1>
0xc1036551      /* mov  (%eax), %eax   ; ret    */
0xc100a7f9      /* xchg   %eax, %esp   ; ret    */
```

Figure 4: Example of an x86 ROP payload (kernel v3.8) used in our ret2dir exploits for elevating privilege.

artificially injected two vulnerabilities that allowed us to corrupt a kernel data or function pointer, and overwrite it with a user-controlled value (marked as "Custom" in Table 2). Note that both flaws are similar to those exploited by the publicly-available exploits. Regarding ARM, the most recent PaX-protected AArch32 kernel that we successfully managed to boot was v3.8.7.

We tested every applicable protection for each exploit. In all cases, the ret2dir versions transfered control *solely* to kernel addresses, bypassing all deployed protections. Figure 3 shows the shellcode we used in x86-64 and AArch32 architectures. The shellcode is *position independent*, so the only change needed in each exploit is to to replace `pkcred` and `ccreds` with the addresses of `prepare_kernel_cred` and `commit_creds`, respectively, as discussed in Section 6.2. By copying the shellcode into a user-space page that has a synonym in the `physmap` area, we can directly execute it from kernel mode by overwriting a kernel code pointer with the `physmap`-resident synonym address of the user-space page. We followed this strategy for all cases in which `physmap` was mapped as executable (corresponding to the entries of Table 2 that contain `SHELLCODE` in the Payload column).

For cases in which `physmap` is non-executable, we substituted the shellcode with a ROP payload that achieves the same purpose. In those cases, the corrupted kernel code pointer is overwritten with the address of a stack pivoting gadget, which brings the kernel's stack pointer to the `physmap` page that is a synonym for the user page that contains the ROP payload. Figure 4 shows an example of an x86 ROP payload used in our exploits. The first gadgets preserve

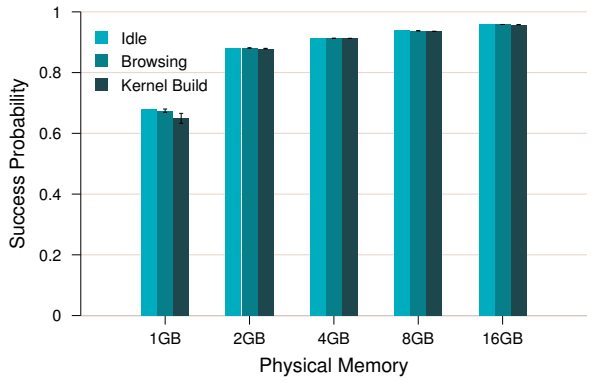


Figure 5: Probability that a selected `physmap` address will point to the exploit payload (successful exploitation) with a single attempt, when using `physmap` spraying, as a function of the available RAM.

the `esp` and `ebp` registers to facilitate reliable continuation (as discussed in Section 6.2). The scratch space can be conveniently located inside the controlled page(s), so the addresses `SCRATCH_SPACE_ADDR1` and `SCRATCH_SPACE_ADDR2` can be easily computed accordingly. The payload then executes essentially the same code as the shellcode to elevate privilege.

7.2 Spraying Performance

In systems without access to `pagemap`, ret2dir attacks have to rely on `physmap` spraying to find a synonym that corresponds to the exploit payload. As discussed in Section 5.2, the probability of randomly selecting a `physmap` address that indeed points to the exploit payload depends on *(i)* the amount of installed RAM, *(ii)* the physical memory load due to competing processes, and *(iii)* the size of the `physmap` area. To assess this probability, we performed a series of experiments under different system configurations and workloads.

Figure 5 shows the probability of successfully selecting a `physmap` address, with a *single* attempt, as a function of the amount of RAM installed in our system; our testbed included a single host armed with two 2.66GHz quad-core Intel Xeon X5500 CPUs and 16GB of RAM, running 64-bit Debian Linux v7. Each bar denotes the average value over 5 repetitions and error bars correspond to 95% confidence intervals. On every repetition we count the percentage of the *maximum* number of `physmap`-resident page frames that we managed to aquire, using the spraying technique (Section 5.2), over the size of `physmap`. We used three different workloads of increasing memory pressure: an idle system, a desktop-like workload with constant browsing activity in multiple tabs (Facebook, Gmail, Twitter, YouTube, and the USENIX website), and a distributed kernel compilation with 16 parallel threads running on 8 CPU cores (`gcc`, `as`, `ld`, `make`). Note that it is necessary to maintain continuous activity in the competing processes so that their working set remains *hot* (worst-case scenario), otherwise the attacking ret2dir processes would easily steal their memory-resident pages.

The probability of success increases with the amount of RAM. For the lowest-memory configuration (1GB), the probability ranges between 65–68%, depending on the workload. This small difference between the idle and the intensive workloads is an indication that despite the continuous activity of the competing processes, the ret2dir processes manage to claim a large amount of memory, as a result of their repeated accesses to all already allocated pages that in essence "lock" them to main memory. For the 2GB configuration the probability jumps to 88%, and reaches 96% for 16GB.

Note that as these experiments were performed on a 64-bit system, `physmap` always mapped all available memory. On 32-bit platforms, in which `physmap` maps only a subset of RAM, the probability of success is even higher. As discussed in Section 5.1, in such cases, the additional memory pressure created by competing processes, which more likely were spawned *before* the ret2dir processes, helps "pushing" ret2dir allocations to the desired zones (`ZONE_NORMAL`, `ZONE_DMA`) that fall within the `physmap` area. Finally, depending on the vulnerability, it is quite common that an unsuccessful attempt will not result in a kernel panic, allowing the attacker to run the exploit multiple times.

8 Defending Against ret2dir Attacks

Restricting access to `/proc/<pid>/pagemap`, or disabling the feature completely (e.g., by compiling the kernel without support for `PROC_PAGE_MONITOR`), is a simple first step that can hinder, but not prevent, ret2dir attacks. In this section, we present an eXclusive Page Frame Ownerwhip (XPFO) scheme for the Linux kernel that provides effective protection with low overhead.

8.1 XPFO Design

XPFO is a thin management layer that enforces *exclusive ownership* of page frames by either the kernel or user-level processes. Specifically, under XPFO, page frames can *never* be assigned to both kernel and user space, unless a kernel component explicitly requests that (e.g., to implement zero-copy buffers [84]).

We opted for a design that does not penalize the performance-critical kernel allocators, at the expense of low additional overhead whenever page frames are allocated to (or reclaimed from) user processes. Recall

that physical memory is allotted to user space using the demand paging and copy-on-write (COW) methods [7], both of which rely on page faults to allocate RAM. Hence, user processes already pay a runtime penalty for executing the page fault handler and performing the necessary bookkeeping. XPFO aligns well with this design philosophy, and increases marginally the management and runtime overhead of user-space page frame allocation. Crucially, the `physmap` area is left untouched, and the slab allocator, as well as kernel components that interface directly with the buddy allocator, continue to get pages that are guaranteed to be physically contiguous and benefit from fast virtual-to-physical address translations, as there are no extra page table walks or modifications.

Whenever a page frame is assigned to a user process, XPFO unmaps its respective synonym from `physmap`, thus breaking unintended aliasing and ensuring that malicious content can no longer be "injected" to kernel space using ret2dir. Likewise, when a user process releases page frames back to the kernel, XPFO maps the corresponding pages back in `physmap` to proactively facilitate dynamic (kernel) memory requests. A key requirement here is to *wipe out* the content of page frames that are returned by (or reclaimed from) user processes, before making them available to the kernel. Otherwise, a non-sanitizing XPFO scheme would be vulnerable to the following attack. A malicious program spawns a child process that uses the techniques presented in Section 5 to map its payload. Since XPFO is in place, the payload is unmapped from `physmap` and cannot be addressed by the kernel. Yet, it will be mapped back once the child process terminates, making it readily available to the malicious program for mounting a ret2dir attack.

8.2 Implementation

We implemented XPFO in the Linux kernel v3.13. Our implementation (~500LOC) keeps the management and runtime overhead to the minimum, by employing a set of optimizations related to TLB handling and page frame cleaning, and handles appropriately *all* cases in which page frames are allocated to (and reclaimed from) user processes. Specifically, XPFO deals with: *(a)* demand paging frames due to previously-requested anonymous and shared memory mappings (`brk`, `mmap`/`mmap2`, `mremap`, `shmat`), *(b)* COW frames (`fork`, `clone`), *(c)* explicitly and implicitly reclaimed frames (`_exit`, `munmap`, `shmdt`), *(d)* swapping (both swapped out and swapped in pages), *(e)* NUMA frame migrations (`migrate_pages`, `move_pages`), and *(f)* huge pages and transparent huge pages.

Handling the above cases is quite challenging. To that end, we first extended the system's page frame data structure (`struct page`) with the following fields: `xpfo_kmcnt` (reference counter), `xpfo_lock` (spinlock) and `xpfo_flags` (32-bit flags field)—`struct page` already contains a flags field, but in 32-bit systems it is quite congested [30]. Notice that although the kernel keeps a `struct page` object for *every* page frame in the system, our change requires only 3MB of additional space per 1GB of RAM (~0.3% overhead). Moreover, out of the 32 bits available in `xpfo_flags`, we only make use of three: "Tainted" (`T`; bit 0), "Zapped" (`Z`; bit 1), and "TLB-shootdown needed" (`S`; bit 2).

Next, we extended the buddy system. Whenever the buddy allocator receives requests for page frames destined to user space (requests with `GFP_USER`, `GFP_HIGHUSER`, or `GFP_HIGHUSER_MOVABLE` set to `gfp_flags`), XPFO unmaps their respective synonyms from `physmap` and asserts `xpfo_flags.T`, indicating that the frames will be allotted to userland and their contents are not trusted anymore. In contrast, for page frames destined to kernel space, XPFO asserts `xpfo_flags.S` (optimization; see below).

Whenever page frames are released to the buddy system, XPFO checks if bit `xpfo_flags.T` was previously asserted. If so, the frame was mapped to user space and needs to be wiped out. After zeroing its contents, XPFO maps it back to `physmap`, resets `xpfo_flags.T`, and asserts `xpfo_flags.Z` (optimization; more on that below). If `xpfo_flags.T` was not asserted, the buddy system reclaimed a frame previously allocated to the kernel itself and no action is necessary (fast-path; no interference with kernel allocations). Note that in 32-bit systems, the above are not performed if the page frame in question comes from `ZONE_HIGHMEM`—this zone contains page frames that are not direct-mapped.

Finally, to achieve complete support of cases *(a)–(f)*, we leverage `kmap`/`kmap_atomic` and `kunmap`/`kunmap_atomic`. These functions are used to temporarily (un)map page frames acquired from `ZONE_HIGHMEM` (see Section 5.1). In 64-bit systems, where the whole RAM is direct-mapped, `kmap`/`kmap_atomic` returns the address of the respective page frame directly from `physmap`, whereas `kunmap`/`kunmap_atomic` is defined as NOP and optimized by the compiler. If XPFO is enabled, all of them are re-defined accordingly.

As user pages are (preferably) allocated from `ZONE_HIGHMEM`, the kernel wraps *all* code related to the cases we consider (e.g., demand paging, COW, swapping) with the above functions. Kernel components that use `kmap` to operate on page frames not related to user processes do exist, and we distinguish these cases using `xpfo_flags.T`. If a page frame is passed to `kmap`/`kmap_atomic` and that bit is asserted, this means that the kernel tries to oper-

ate on a frame assigned to user space via its kernel-resident synonym (e.g., to read its contents for swapping it out), and thus is temporarily mapped back in physmap. Likewise, in kunmap/kunmap_atomic, page frames with xpfo_flags.T asserted are unmapped. Note that in 32-bit systems, the XPFO logic is executed on kmap routines only for direct-mapped page frames (see Table 1). xpfo_lock and xpfo_kmcnt are used for handling recursive or concurrent invocations of kmap/kmap_atomic and kunmap/kunmap_atomic with the same page frame.

Optimizations: The overhead of XPFO stems mainly from two factors: *(i)* sanitizing the content of reclaimed pages, and *(ii)* TLB shootdown and flushing (necessary since we modify the kernel page table). We employ three optimizations to keep that overhead to the minimum. As full TLB flushes result in prohibitive slowdowns [53], in architectures that support single TLB entry invalidation, XPFO selectively evicts only those entries that correspond to synonyms in physmap that are unmapped; in x86/x86-64 this is done with the INVLPG instruction.

In systems with multi-core CPUs, XPFO must take into consideration TLB coherency issues. Specifically, we have to perform a TLB shootdown whenever a page frame previously assigned to the kernel itself is mapped to user space. XPFO extends the buddy system to use xpfo_flags.S for this purpose. If that flag is asserted when a page frame is alloted to user space, XPFO invalidates the TLB entries that correspond to the synonym of that frame in physmap, in *every* CPU core, by sending IPI interrupts to cascade TLB updates. In all other cases (i.e., page frames passed from one process to another, reclaimed page frames from user processes that are later on alloted to the kernel, and page frames allocated to the kernel, reclaimed, and subsequently allocated to the kernel again), XPFO performs only local TLB invalidations.

To alleviate the impact of page sanitization, we exploit the fact that page frames previously mapped to user space, and in turn reclaimed by the buddy system, have xpfo_flags.Z asserted. We extended clear_page to check xpfo_flags.Z and avoid clearing the frame if the bit is asserted. This optimization avoids zeroing a page frame twice, in case it was first reclaimed by a user process and then subsequently allocated to a kernel path that required a clean page—clear_page is invoked by every kernel path that requires a zero-filled page frame.

Limitations: XPFO provides protection against ret2dir attacks, by braking the unintended address space sharing between different security contexts. However, it does not prevent generic forms of data sharing between kernel and user space, such as user-controlled content pushed to kernel space via I/O buffers, the page cache, or through system objects like pipes and message queues.

Benchmark	Metric	Original	XPFO (%Overhead)
Apache	Req/s	17636.30	17456.47 **(%1.02)**
NGINX	Req/s	16626.05	16186.91 **(%2.64)**
PostgreSQL	Trans/s	135.01	134.62 **(%0.29)**
Kbuild	sec	67.98	69.66 **(%2.47)**
Kextract	sec	12.94	13.10 **(%1.24)**
GnuPG	sec	13.61	13.72 **(%0.80)**
OpenSSL	Sign/s	504.50	503.57 **(%0.18)**
PyBench	ms	3017.00	3025.00 **(%0.26)**
PHPBench	Score	71111.00	70979.00 **(%0.18)**
IOzone	MB/s	70.12	69.43 **(%0.98)**
tiobench	MB/s	0.82	0.81 **(%1.22)**
dbench	MB/s	20.00	19.76 **(%1.20)**
PostMark	Trans/s	411.00	399.00 **(%2.91)**

Table 3: XPFO performance evaluation results using macro-benchmarks (upper part) and micro-benchmarks (lower part) from the Phoronix Test Suite.

8.3 Evaluation

To evaluate the effectiveness of the proposed protection scheme, we used the ret2dir versions of the real-world exploits presented in Section 7.1. We back-ported our XPFO patch to each of the six kernel versions used in our previous evaluation (see Table 2), and tested again our ret2dir exploits when XPFO was enabled. In all cases, XPFO prevented the exploitation attempt.

To assess the performance overhead of XPFO, we used kernel v3.13, and a collection of macro-benchmarks and micro-benchmarks from the Phoronix Test Suite (PTS) [82]. PTS puts together *standard* system tests, like apachebench, pgbench, kernel build, and IOzone, typically used by kernel developers to track performance regressions. Our testbed was the same with the one used in Section 7.2; Table 3 summarizes our findings. Overall, XPFO introduces a minimal (negligible in most cases) overhead, ranging between 0.18–2.91%.

9 Conclusion

We have presented ret2dir, a novel kernel exploitation technique that takes advantage of direct-mapped physical memory regions to bypass existing protections against ret2usr attacks. To improve kernel isolation, we designed and implemented XPFO, an exclusive page frame ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory. The results of our experimental evaluation demonstrate that XPFO offers effective protection with negligible runtime overhead.

Availability

Our prototype implementation of XPFO and all modified ret2dir exploits are available at: http://www.cs.columbia.edu/~vpk/research/ret2dir/

Acknowledgments

This work was supported by DARPA and the US Air Force through Contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, respectively, with additional support from Intel Corp. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, or Intel.

References

[1] `pagemap`, from the userspace perspective, December 2008. `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`.

[2] ACCETTA, M. J., BARON, R. V., BOLOSKY, W. J., GOLUB, D. B., RASHID, R. F., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of USENIX Summer* (1986), pp. 93–113.

[3] ARGYROUDIS, P. Binding the Daemon: FreeBSD Kernel Stack and Heap Exploitation. In *Black Hat USA* (2010).

[4] ARM® ARCHITECTURE REFERENCE MANUAL. ARM®v7-A and ARM®v7-R edition. Tech. rep., Advanced RISC Machine (ARM), July 2012.

[5] BEN HAYAK. The Kernel is calling a zero(day) pointer - CVE-2013-5065 - Ring Ring, December 2013. `http://blog.spiderlabs.com/2013/12/the-kernel-is-calling-a-zeroday-pointer-cve-2013-5065-ring-ring.html`.

[6] BONWICK, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer* (1994), pp. 87–98.

[7] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 3rd ed. 2005, ch. Memory Management, pp. 294–350.

[8] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-Oriented Programming without Returns. In *Proc. of CCS* (2010), pp. 559–572.

[9] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. of APsys* (2011), pp. 51–55.

[10] COMMON VULNERABILITIES AND EXPOSURES. CVE-2005-0736, March 2005.

[11] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-1527, May 2009.

[12] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-2698, August 2009.

[13] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-3002, August 2009.

[14] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-3234, September 2009.

[15] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-3547, October 2009.

[16] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-2959, August 2010.

[17] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-3437, September 2010.

[18] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-3904, October 2010.

[19] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-4073, October 2010.

[20] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-4347, November 2010.

[21] COMMON VULNERABILITIES AND EXPOSURES. CVE-2012-0946, February 2012.

[22] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-0268, December 2013.

[23] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-1828, February 2013.

[24] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-2094, February 2013.

[25] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-2852, April 2013.

[26] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-2892, August 2013.

[27] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-4343, June 2013.

[28] CORBET, J. Virtual Memory I: the problem, March 2004. `http://lwn.net/Articles/75174/`.

[29] CORBET, J. An updated guide to `debugfs`, May 2009. `http://lwn.net/Articles/334546/`.

[30] CORBET, J. How many page flags do we really have?, June 2009. `http://lwn.net/Articles/335768/`.

[31] CORBET, J. Supervisor mode access prevention, October 2012. `http://lwn.net/Articles/517475/`.

[32] CORBET, J., KROAH-HARTMAN, G., AND MCPHERSON, A. Linux Kernel Development. Tech. rep., Linux Foundation, September 2013.

[33] DING, Y., WEI, T., WANG, T., LIANG, Z., AND ZOU, W. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *Proc. of ACSAC* (2010), pp. 327–336.

[34] DISTROWATCH. Put the fun back into computing. Use Linux, BSD., November 2013. `http://distrowatch.com`.

[35] EDGE, J. Kernel address space layout randomization, October 2013. `http://lwn.net/Articles/569635/`.

[36] EXPLOIT DATABASE. EBD-131, December 2003.

[37] EXPLOIT DATABASE. EBD-16835, September 2009.

[38] EXPLOIT DATABASE. EBD-14814, August 2010.

[39] EXPLOIT DATABASE. EBD-15150, September 2010.

[40] EXPLOIT DATABASE. EBD-15285, October 2010.

[41] EXPLOIT DATABASE. EBD-15916, January 2011.

[42] EXPLOIT DATABASE. EBD-17391, June 2011.

[43] EXPLOIT DATABASE. EBD-17787, September 2011.

[44] EXPLOIT DATABASE. EBD-20201, August 2012.

[45] EXPLOIT DATABASE. EBD-24555, February 2013.

[46] GEORGE, V., PIAZZA, T., AND JIANG, H. Technology Insight: Intel©Next Generation Microarchitecture Codename Ivy Bridge, September 2011. `http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf`.

[47] GOOGLE. Android, November 2013. `http://www.android.com`.

[48] GOOGLE. Chromium OS, November 2013. `http://www.chromium.org/chromium-os`.

[49] HARDY, N. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev. 22* (October 1988), 36–38.

[50] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. MINIX 3: A Highly Reliable, Self-

Repairing Operating System. *SIGOPS Oper. Syst. Rev. 40*, 3 (July 2006), 80–89.

[51] HUND, R., HOLZ, T., AND FREILING, F. C. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proc. of USENIX Sec* (2009), pp. 384–398.

[52] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (April 2007), 37–49.

[53] INGO MOLNAR. 4G/4G split on x86, 64 GB RAM (and more) support, July 2003. http://lwn.net/Articles/39283/.

[54] INTEL® 64 AND IA-32 ARCHITECTURES SOFTWARE DEVELOPER'S MANUAL. Instruction Set Extensions Programming Reference. Tech. rep., Intel Corporation, January 2013.

[55] INTEL® 64 AND IA-32 ARCHITECTURES SOFTWARE DEVELOPER'S MANUAL. System Programming Guide, Part 1. Tech. rep., Intel Corporation, September 2013.

[56] JANA, S., AND SHMATIKOV, V. Memento: Learning Secrets from Process Footprints. In *Proc. of IEEE S&P* (2012), pp. 143–157.

[57] KEMERLIS, V. P., PORTOKALIDIS, G., AND KEROMYTIS, A. D. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proc. of USENIX Sec* (2012), pp. 459–474.

[58] KILLIAN, T. J. Processes as Files. In *Proc. of USENIX Summer* (1984), pp. 203–207.

[59] KING, R. Kernel Memory Layout on ARM Linux, November 2005. https://www.kernel.org/doc/Documentation/arm/memory.txt.

[60] KLEEN, A. Memory Layout on amd64 Linux, July 2004. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[61] KNOWLTON, K. C. A fast storage allocator. *Commun. ACM 8*, 10 (October 1965), 623–624.

[62] KOLDINGER, E. J., CHASE, J. S., AND EGGERS, S. J. Architecture Support for Single Address Space Operating Systems. In *Proc. of ASPLOS* (1992), pp. 175–186.

[63] KURMUS, A., TARTLER, R., DORNEANU, D., HEINLOTH, B., ROTHBERG, V., RUPRECHT, A., SCHRÖDER-PREIKSCHAT, W., LOHMANN, D., AND KAPITZA, R. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proc. of NDSS* (2013).

[64] LAMETER, C. Generic Virtual Memmap support for SPARSEMEM v3, April 2007. http://lwn.net/Articles/229670/.

[65] LIAKH, S. NX protection for kernel data, July 2009. http://lwn.net/Articles/342266/.

[66] LIEDTKE, J. On μ-Kernel Construction. In *Proc. of SOSP* (1984), pp. 237–250.

[67] LOVE, R. *Linux Kernel Development*, 2nd ed. 2005, ch. Memory Management, pp. 181–208.

[68] MARINAS, C. arm64: Distinguish between user and kernel XN bits, November 2012. https://forums.grsecurity.net/viewtopic.php?f=7&t=3292.

[69] MARINAS, C. Memory Layout on AArch64 Linux, February 2012. https://www.kernel.org/doc/Documentation/arm64/memory.txt.

[70] MCDOUGALL, R., AND MAURO, J. *Solaris Internals*, 2nd ed. 2006, ch. File System Framework, pp. 710–710.

[71] MOKB. Broadcom Wireless Driver Probe Response SSID Overflow, November 2006.

[72] MOZILLA. Firefox OS, November 2013. https://www.mozilla.org/en-US/firefox/os/.

[73] NATIONAL VULNERABILITY DATABASE. Kernel Vulnerabilities, November 2013. http://goo.gl/GJpw0b.

[74] NILS AND JON. Polishing Chrome for Fun and Profit, August 2013. http://goo.gl/b5hmjj.

[75] OFFENSIVE SECURITY. The Exploit Database, November 2013. http://www.exploit-db.com.

[76] ORMANDY, T., AND TINNES, J. Linux ASLR Curiosities. In *CanSecWest* (2009).

[77] PAX. Homepage of The PaX Team, November 2013. http://pax.grsecurity.net.

[78] PAX TEAM. UDEREF/i386, April 2007. http://grsecurity.net/~spender/uderef.txt.

[79] PAX TEAM. UDEREF/amd64, April 2010. http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html.

[80] PAX TEAM. Better kernels with GCC plugins, October 2011. http://lwn.net/Articles/461811/.

[81] PERLA, E., AND OLDANI, M. *A Guide To Kernel Exploitation: Attacking the Core*. 2010, ch. Stairway to Successful Kernel Exploitation, pp. 47–99.

[82] PTS. Phoronix Test Suite, February 2014. http://www.phoronix-test-suite.com.

[83] RAMON DE CARVALHO VALLE & PACKET STORM. sock_sendpage() NULL pointer dereference (PPC/PPC64 exploit), September 2009. http://packetstormsecurity.org/files/81212/Linux-sock_sendpage-NULL-Pointer-Dereference.html.

[84] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC* (2012), pp. 101–112.

[85] ROSENBERG, D. Owned Over Amateur Radio: Remote Kernel Exploitation in 2011. In *Proc. of DEF CON®* (2011).

[86] SECURITYFOCUS. Linux Kernel 'perf_counter_open()' Local Buffer Overflow Vulnerability, September 2009.

[87] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of CCS* (2007), pp. 552–61.

[88] SINAN EREN. Smashing The Kernel Stack For Fun And Profit. *Phrack 6*, 60 (December 2002).

[89] SPENGLER, B. Enlightenment Linux Kernel Exploitation Framework, July 2013. https://grsecurity.net/~spender/exploits/enlightenment.tgz.

[90] SPENGLER, B. Recent ARM security improvements, February 2013. https://forums.grsecurity.net/viewtopic.php?f=7&t=3292.

[91] SQRKKYU, AND TWZI. Attacking the Core: Kernel Exploiting Notes. *Phrack 6*, 64 (May 2007).

[92] VAN DE VEN, A. Debug option to write-protect rodata: the write protect logic and config option, November 2005. http://lkml.indiana.edu/hypermail/linux/kernel/0511.0/2165.html.

[93] VAN DE VEN, A. Add -fstack-protector support to the kernel, July 2006. http://lwn.net/Articles/193307/.

[94] VAN DE VEN, A. x86: add code to dump the (kernel) page tables for visual inspection, February 2008. http://lwn.net/Articles/267837/.

[95] YU, F. Enable/Disable Supervisor Mode Execution Protection, May 2011. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=de5397ad5b9ad22e2401c4dacdf1bb3b19c05679.

JIGSAW : Protecting Resource Access by Inferring Programmer Expectations

Hayawardh Vijayakumar[1], Xinyang Ge[1], Mathias Payer[2], and Trent Jaeger[1]

[1]SIIS Laboratory, Department of CSE
The Pennsylvania State University
{hvijay,xxg113,tjaeger}@cse.psu.edu
[2]EECS Department
University of California Berkeley
mathias.payer@nebelwelt.net

Abstract

Processes retrieve a variety of resources, such as files, from the operating system to function. However, securely accessing resources has proven to be a challenging task, accounting for 10-15% of vulnerabilities reported each year. Current defenses address only a subset of these vulnerabilities in ad-hoc and incomplete ways. In this paper, we provide a comprehensive defense against vulnerabilities during resource access. First, we identify a fundamental reason that resource access vulnerabilities exist – a mismatch between programmer expectations and the actual environment the program runs in. To address such mismatches, we propose JIGSAW, a system that can automatically derive programmer expectations and enforce it on the deployment. JIGSAW constructs programmer expectations as a *name flow graph*, which represents the data flows from the inputs used to construct file pathnames to the retrieval of system resources using those pathnames. We find that whether a program makes any attempt to filter such flows implies expectations about the threats the programmer expects during resource retrieval, the enabling JIGSAW to enforce those expectations. We evaluated JIGSAW on widely-used programs and found that programmers have many implicit expectations. These mismatches led us to discover two previously-unknown vulnerabilities and a default misconfiguration in the Apache webserver. JIGSAW enforces program expectations for approximately 5% overhead for Apache webservers, thus eliminating vulnerabilities due to resource access efficiently and in a principled manner.

1 Introduction

Processes retrieve a variety of *resources* from the operating system to function. A resource is any abstraction that the system call API of an operating system (OS) offers to a process (apart from a process itself). Examples of resources are files (configuration, data, or log files), network ports, or interprocess communication channels (IPCs) such as sockets and shared memory. Such OS abstractions free the programmer from having to know details of the underlying hardware and allow her to write portable code. Conceptually, resource access is a procedure that takes as input the name (e.g., filename) and namespace bindings (e.g., directories or symbolic links), and returns the resource (e.g., file) as output to the process.

Securely accessing resources has proven to be a challenging task because of adversarial control of the inputs to resource access. Adversaries may control the input name or a binding to direct victim processes to unsafe resources. In the well-known *time-of-check to time-of-use* (TOCTTOU) attack [6], an adversary exploits the non-atomicity between check operations (e.g., access) and use operations (e.g., open) to redirect the victim to resources of the adversary's choosing. Other attacks are link following, untrusted search paths, Trojan-horse library loads, and directory traversal. These attacks are collectively referred to as *resource access attacks* [40]. 10-15% of the vulnerabilities reported each year in the CVE database [12] are due to programs not defending themselves against these attacks.

Current defenses against such vulnerabilities in conventional OSes are ad-hoc and fundamentally limited. First, traditional access control is too coarse-grained to prevent resource access vulnerabilities. A process may have legitimate access to both low-integrity adversarial files and high-integrity library files; however, the resource access to load libraries should not access adversary files (and vice versa). Traditional access control does not differentiate between these resource accesses, and thus cannot protect programs. Second, defenses in the research literature require either system or program modifications. System defenses have been mainly limited to TOCTTOU [11, 13, 25–27, 35–38, 44] and link following [10], or require programs to be written to new APIs [19,29,34,42]. However, system defenses are fund-

mentally limited because they do not have sufficient information about programs [8], and new APIs do not protect existing programs.

This paper presents an approach to automatically protect programs that use the current system call API from resource access vulnerabilities. We make the following observations: first, we find that a fundamental cause for resource access vulnerabilities is programmer expectations not being satisfied during the program's system deployment. For example, during a particular resource access, a programmer might expect to fetch a resource that is inaccessible to an adversary (e.g., a log file in `/var/log`) and thus not add defensive code checks, called *filters*, to protect against adversarial control of names and bindings. However, this expectation may not be consistent with the view of the OS distributors (e.g., Red Hat, Ubuntu) who actually frame the access control policy. Thus, if permissions to `/var/log` allow adversary access (e.g., through a world-writable directory), adversaries can compromise the victim program. Our second insight is that we can *automatically* infer if the programmer expected adversarial control at a particular resource access or not, without requiring any annotations or changes to the program. We do this by detecting the presence of name and binding filters that programmers place in the program.

In this paper, we develop JIGSAW, the first system to provide automated protection for programs during resource access without requiring additional programmer effort. JIGSAW infers programmer expectations and enforces these on the deployment[1]. JIGSAW is based on two conceptually simple invariants – that the system deployment's attack surface be a subset of the programmer's expected attack surface, and that resource accesses not result in confused deputy attacks [16]. These invariants, if correctly evaluated and enforced, can theoretically provide complete program protection during resource access. JIGSAW operates in two phases. In the first phase, it mines programmer expectations by detecting the presence of filters in program code. Using these filters, JIGSAW constructs a novel representation, called the *name flow graph*, from which the programmer's expected attack surface is derived. We show that anomalous cases in the name flow graph can be used to detect vulnerabilities to resource access attacks. In the second phase, JIGSAW enforces the invariants by leveraging the open-source Process Firewall of Vijayakumar *et al.* [40], a Linux kernel module that (i) knows about deployment attack surface using the system's adversary accessibility, and (ii) introspects into the program to identify the resource access and to enforce its expectations as determined in the first phase.

[1]Informally, JIGSAW enables "fitting" the programmer's expectations on to its deployment.

We evaluate our technique by hardening widely-used programs against resource access attacks. Our results show that in general, programmers have many implicit expectations during resource access. For example, in the Apache web server, we found 65% of all resource accesses are implicitly expected not to be under adversary control. However, this is not conveyed to OS distributors in any form, and may result in vulnerabilities. JIGSAW can be use to detect such vulnerabilities, and we did find two *previously-unknown* vulnerabilities and a default misconfiguration. However, the key feature of JIGSAW is that it protects program vulnerabilities during resource access whenever there is a discrepancy between the programmers' inferred expectations and the system configuration, without the need to modify the program or the system's access control policies. We also find that the Process Firewall can enforce such protection to block resource access vulnerabilities whilst allowing legitimate functionality for a modest performance overhead of approximately 5%. An automated analysis as presented in this paper can thus enforce efficient protection for programs during resource access at runtime.

In summary, we make the following contributions.

- We precisely define resource access vulnerabilities and show how they occur due to a mismatch in expectations between the programmer, the OS distributor, and the administrator,
- We propose two invariants that, if evaluated and enforced correctly, can theoretically provide complete program protection during resource access,
- We develop JIGSAW, an automated approach that uses the invariants to protect programs during resource access by inferring programmer expectation using the novel abstraction of a name flow graph, and
- We evaluate our approach on widely-used programs, showing how programmers have many implicit expectations, as demonstrated by our discovery of two previously-unknown vulnerabilities and a default misconfiguration in our deployment of the Apache web server. Further, we show that we can produce rules to enforce these implicit assumptions efficiently using the Process Firewall on any program deployment.

2 Problem definition

In this section, we first give a precise definition of when a resource access is vulnerable. This definition classifies vulnerabilities into two broad categories. We then identify the fundamental cause for each of these vulnerability categories – mismatch between programmer expectation and system deployment, and difficulty in writing proper defensive code.

```
01  conf = open("httpd.conf");
02  log_file = read(conf);
03  socket = bind(port 80);
04  open(log_file, O_CREAT);          // File Squat
05  loop {
06    html_page = recv(socket);
07    strip(html_page, "../");        // Directory Traversal
08    stat(html_page not symlink);    // TOCTTOU Race
09    open(html_page, O_RDONLY);      // TOCTTOU Race, Symlink
10    write(client_socket, "200 OK");
11    log("200 OK to client")
12  }
```

Figure 1: Motivating example of resource access vulnerabilities using a typical processing cycle of a web server.

2.1 Resource Access Attacks

A resource access occurs when a program uses a *name* to resolve a *resource* using namespace *bindings*. That is, the inputs to the resource access are the name and the bindings, and the output is the final resource. Figure 1 shows example webserver code that we use throughout the paper: the webserver starts up and accesses its configuration file (line 2), from which it gets the location of its log file. It then binds a socket on port 80 (line 3), opens the log file (line 4), and waits for client requests. When a client connects, it receives the HTTP request (line 6), uses this name to fetch the HTML file (line 9). Finally, it writes the status code to its log file (line 11).

Let us examine some possible resource access vulnerabilities. Consider line 6. Here, the program receives a HTTP request from the client, and serves the page to the client. The client can supply a name such as `../../etc/passwd`, and if the server does not properly sanitize the name (which it attempts to do in line 7), the client is served the password file on the server. This is a *directory traversal* vulnerability. Next, consider the check the server makes in line 8. Here, the server checks that the HTML file is not a symbolic link. The reason for this is that in many deployments (e.g., a university web server serving student web pages), the web page is controlled by an adversary (i.e., student). The server attempts to prevent a symbolic link vulnerability, where a student links her web page to the password file. However, a race condition between the check in line 8 and the use in line 9, leads to a *link following* vulnerability exploiting a *TOCTTOU race condition*.

To see how such vulnerabilities can be broadly classified, we introduce adversary accessibility to resources and adversarial control of resource access. We then define when resource accesses are vulnerable, and use this to derive a classification for vulnerabilities.

Adversary accessible resources. An *adversary-accessible resource* is one that an adversary has permissions to access (read for secrecy attacks, write for integrity attacks) under the system's access control policy. The complement set is the set of *adversary-inaccessible* resources.

Expected/Safe Resource	*Malicious/Unsafe Resource*	*Vulnerability Class*
Adversary-Inaccessible (Hi) Resource	*Adversary-Accessible (Lo) Resource*	**Unexpected Attack Surface** Untrusted Search Path File/IPC Squat PHP File Inclusion
Adversary-Accessible (Lo) Resource	*Adversary-Inaccessible (Hi) Resource*	**Confused Deputy** Link Following Directory Traversal TOCTTOU races

Table 1: Adversaries control resource access to direct victims to adversary-accessible resources when the victim expected an adversary inaccessible resource and vice-versa.

Adversary control of resource access. An adversary controls the resource access by controlling its inputs (the name or a binding). An adversary controls a binding if she uses her write permissions in a directory to create a binding [39]. An adversary controls a name if there is an explicit data flow[2] from an adversary-accessible resource to the name used in resource access. The adversary needs write permissions to these resources to control names.

The directory traversal vulnerability above relies on the adversary's ability to control the name used in resource access. The link following vulnerability relies on the adversary's ability to control the binding (creating a symbolic link).

Resource Access Vulnerability. A resource access is vulnerable, i.e., a resource access attack is successful, when an adversary uses her control of inputs to resource access (the name or a binding) to direct a victim to an adversary-accessible output resource when the victim expected an adversary-inaccessible resource (and vice versa).

On the one hand, the adversary can use control to direct the victim to an adversary-inaccessible resource when the program expects an adversary-accessible resource. The directory traversal and link following vulnerabilities of the classical *confused deputy* [16] (Row 2 in Table 1). On the other hand, the adversary can direct the victim to an adversary-accessible resource when the program expects an adversary-inaccessible resource. Trojan-horse libraries is an example vulnerability of this type. We call these *unexpected attack surface* vulnerabilities (Row 1 in Table 1), as they occur because the programmer is not expecting adversary control at these resource accesses. Table 1 classifies all resource access vulnerabilities into these two types.

2.2 Causes for Resource Access Vulnerabilities

We identify two causes for resource access vulnerabilities – one for each category in Table 1. The first cause is a mismatch in expectations of adversary control of names and bindings between the program and the deployment.

[2] Attacks involving names require the adversary to inject sequences of `../` or unicode characters. Thus, explicit data flow is necessary; just implicit data flow is insufficient.

Consider Figure 2 that describes resource accesses from the web server example in Figure 1. Here, the programmer expects the resource access of the HTML file to be under adversary control, and to combat this, adds a name filter [3] from the TCP socket (stripping `../`) as well as a binding filter (check for a link). The programmer did not expect the log file's resource access to be adversary-controlled, and therefore did not add any filters. However, due to a misconfiguration, this programmer expectation is not satisfied in the deployment configuration, causing a resource access vulnerability.

In general, resource access vulnerabilities are very challenging to eliminate because they involve multiple disconnected parties. First, programmers write code assuming that a certain subset of resource accesses are under adversarial control. Resource access checks cause overhead, so the programmer generally tries to minimize the number of checks, thereby motivating fewer filters. Second, there are OS distributors who define access control policies, thereby determining adversarial control of resource accesses. However, these OS distributors have little or no information about the assumptions the programmer has made about adversarial control, resulting in a set of possible mismatches. Finally, there are administrators who deploy the program on a concrete system. The configuration specifies the location of various resources such as log files. Thus, the administrator's configuration too may not match the programmer's expectation.

The second cause for resource access vulnerabilities is where the programmer does expect adversary-controlled resource access, but the filter may be insufficient to protect the program. Note that when a program encounters an adversary-controlled resource access, the only valid resource is an adversary-accessible resource; otherwise, the program is victim to a confused deputy vulnerability. Thus, the program needs to defend itself by filtering improper requests leading to a confused deputy. However, both name and binding filters are difficult to get right due to difficulty in string parsing [4] and inherent race conditions in the system call API [8] (e.g., lines 8, 9 in Figure 1).

In summary, the two causes of resource access vulnerabilities are: *(a)* unexpected adversarial control of resource access, and *(b)* improper filtering of resource access when adversary control of resource access is expected. These causes correspond to Rows 1 and 2 in Table 1 respectively. With these two causes in mind, we proceed to a model that precisely describes our solution.

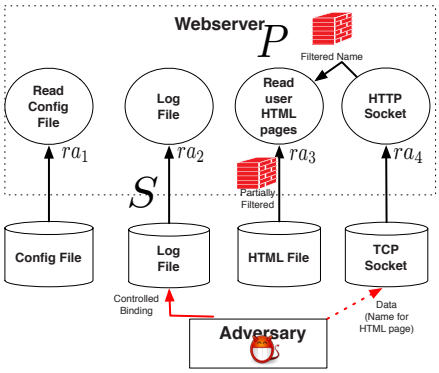


Figure 2: Demonstrating a mismatch between the expected and deployed attack surfaces.

3 Model and Solution Overview

In this section, we provide two invariants that directly address the two causes for resource access vulnerabilities outlined above.

Consider the set of all resource accesses *RA* made by a program[4]. A resource access happens when a system call resolves a resource using a name and namespace bindings. The program has code to filter the names and bindings used during certain resource accesses (e.g., ra_3 in Figure 2). From this knowledge, we show in Section 5 how to derive *P*, the set of resource accesses that a programmer expects to be under adversarial control. This set *P* is the *expected resource access attack surface*, or simply, the expected attack surface.

Now, assume that the program is deployed and run on a system. A subset of the resource accesses made by the program is adversary-controlled in the deployment. Let *Y* be the deployment's access control policy. Let *S* be the set of resource accesses that are adversary-controlled under *Y* (ra_2 in Figure 2). This set *S* defines the *deployment resource access attack surface*, or simply, the deployment attack surface.

Given *Y*, the expected attack surface *P* is *safe* for the deployment *S* if $S \subseteq P$, i.e., if all resource accesses in the deployment attack surface are part of the program's expected attack surface. Intuitively, this means that the program has filters to protect itself whenever a resource access is adversary-controlled. If *r* is the resource access under consideration, then, the invariant stated in propositional logic blocking unexpected adversary is:

$$\textbf{Invariant: Unexpected Adversary Control}(r): (r \in S) \rightarrow (r \in P) \quad (1)$$

If this safety invariant is enforced, resource access vul-

[3] A filter is a check in code that allows only a subset of names, bindings and resources through.

[4] The representation used to identify a resource access is implementation-dependent. In our implementation, we use program stacks at the time of the resource access system call.

nerabilities are eliminated where programs do not expect adversary control. Therefore, vulnerabilities are only possible where programs expect adversary control.

Now that we are dealing with an adversary-controlled resource access ($\in S$) that is also expected ($\in P$), the only valid resource is an adversary-accessible resource; otherwise, the program would be victim to a confused deputy attack. We say that resource accesses in P are protected from a confused deputy vulnerability if, when the resource access is adversary-controlled (i.e., $\in S$), it does not accept adversary-inaccessible resources. Let R be the set of resource accesses that retrieve adversary-accessible resources (as defined under Y). Then, a resource access r is protected from confused deputy vulnerabilities if the following invariant stated in proposition logic holds:

$$\textbf{Invariant: Confused Deputy}(r): \quad (r \in S) \rightarrow (r \in R) \tag{2}$$

Once these two rules are enforced, the only resources that are allowed are adversary-accessible resources where programs expect adversary control. Problems occur if the program does not properly handle this adversary-accessible resource. For example, if it does not filter data read from this resource properly, memory corruption vulnerabilities might result. Such vulnerabilities that occur in spite of fetching the expected resource are not within the scope of this work.

Let us examine how the rules above stop the vulnerability classes in Table 1. Consider vulnerability in Row 2, where the victim expects an adversary inaccessible resource (high integrity or secrecy), but ends up with an adversary-accessible (low integrity or secrecy) resource. The typical case is an untrusted search path where the program expects to load a high-integrity library, but searches for libraries in insecure paths due to programmer oversight or insecure environment variables, and ends up with a Trojan horse low-integrity library. Here, since the programmer does not expect a low-integrity library, she does not place a binding (or name) filter. Thus, we will infer that this resource access is not part of the expected attack surface ($\notin P$), and Invariant 1 above will stop the vulnerability if this resource access is controlled in any way (binding or name) by an adversary ($\in S$). The other vulnerability classes in this category are blocked similarly. Next, consider vulnerabilities in Row 1. Here, the victim expects an adversary-accessible resource (low integrity or secrecy), but ends up with an adversary inaccessible resource (high integrity or secrecy). In a link following vulnerability, the adversary creates a symbolic link to a high-secrecy or high-integrity file, such as the password file. Thus, the adversary uses her control of bindings ($\in S$) to direct the victim to an adversary-inaccessible resource ($\notin R$). In a directory traversal vulnerability, the adversary uses her control of the name to supply sequences of `../` to direct the victim to a high-secrecy or high-integrity file. In both cases, Invariant 2 will block the vulnerability since the adversary controls the resource access ($\in S$) through the name or binding, but the resource is adversary inaccessible ($\notin R$).

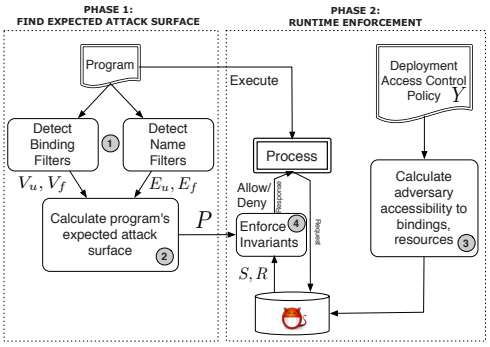


Figure 3: Overview of the design of our system.

4 JIGSAW Approach Overview

Figure 3 shows an outline of the design of JIGSAW. JIGSAW has two phases. In the first phase, JIGSAW calculates P, the expected attack surface. Finding P requires inferring programmer expectations. To infer programmer expectations, we propose an intuitive heuristic – if the programmer expects adversary control at a resource access, she will place filters in code to handle such control. Given the program, we perform a black-box analysis to detect the existence of any binding and name filtering separately (Step 1 in Figure 3), and use this information to calculate the program's expected attack surface (Step 2).

In the second phase, JIGSAW enforces Invariants 1 and 2 above by determining S, the deployment attack surface, and R, the set of adversary-accessible resources. The deployment's access control policy Y determines which resources and bindings are adversary-accessible. We leverage existing techniques to calculate adversary accessibility given Y [10, 17, 41] (Step 3). At runtime, if an adversary-accessible resource is used, that resource access is in R. If the name is read from an adversary-accessible resource or the binding used in resolving that name is adversary accessible, then that resource access is in S. Finally, we need to enforce Invariants 1 and 2 for individual resource accesses (Step 4). Any enforcement mechanism that applies distinct access control rules per individual resource access system call would be suitable. In our prototype implementation we leverage the open-source Process Firewall [40] which enables us to support binary-only programs (i.e., our prototype implementation does not rely on source code access).

5 Phase 1: Find Expected Attack Surfaces

The first step is to determine the expected attack surface P for a program. We do this in two parts. First, we propose a heuristic that implies the expectations of programmers with respect to the adversary control of the inputs to resource access and introduce the abstraction of a *name flow graph* to model these expectations and enable the detection of missing filters (Sections 5.1 to 5.3). Next, we outline how one can use dynamic analysis methods to build name flow graphs by accounting for adversary control of names and bindings (Sections 5.4 and 5.5).

5.1 Resource Access Expectations

Determining P requires knowledge of the programmers' expectations – whether the programmers expected the resource access to be under adversary control or not. The most precise solution to this problem is to ask each programmer to specify her expectation. Unfortunately, such annotations do not exist currently. As an alternative, we use the presence of code filters to infer programmer expectation. We use the following heuristic:

Heuristic. *If a programmer expects adversarial control of a resource access, she will add code filters to protect the program from adversarial control.*

Thus, the way we infer if a programmer expects an adversary-controlled resource access is by detecting if she adds *any* code to filter such adversarial control. An adversary controls a resource access by controlling either the name or a binding used in the resource access. Thus, we need to detect whether a program filters names and bindings separately.

Before presenting exactly how we detect filtering, we will introduce the concept of a *name flow graph* for a program, which we will use to derive the expected attack surface P given knowledge of filtered resource accesses.

5.2 Name Flow Graph

We introduce the abstraction of a name flow graph, which represents the data flow of name values among resource accesses in the program annotated with the knowledge of whether names and/or bindings are filtered each individual resource access. Using this name flow graph, we will show that we can compute resources accesses that are missing filters automatically. A name flow graph $G_n = (V, E)$ is a graph where the resource accesses are nodes and each edge $(a, b) \in E$ represents whether *there exists* a data flow in the program between the data of any of the resources retrieved at the access at node a and any of the name variables used in an access at node b. We refer to these edges as *name flows*.

Further, $V = V_f \cup V_u$ where V_f is the set of resource accesses that filter bindings and V_u the set of vertices that do not. Similarly, $E = E_f \cup E_u$, where E_f is the set of name flows that are filtered, and E_u the set that is not. That is, a name flow graph is a data-flow graph that captures the flow of names and is annotated with information about filters. The meaning of filtering for names and bindings is described in Sections 5.4 and 5.5, respectively.

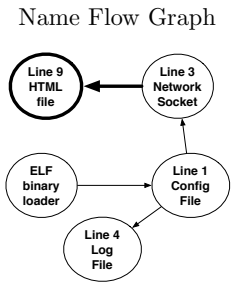


Figure 4: Name flow graph for the example in Figure 1.

The name flow graph for our web server in Figure 1 is shown in Figure 4. Its nodes are resource accesses and edges connect two resource accesses if the data read at the source resource access may affect the name used at the target resource access. The bold nodes are those that filter bindings, whereas the bold edges are those that filter names.

The name flow graph determines P, the expected attack surface. According to our heuristic in Section 5, a resource access is part of the expected attack surface if a programmer places both name and binding filters on the resource access to handle adversarial control. However, not all name flows need be filtered – only name flows originating from other resource accesses also under adversarial control must be. Since this definition is transitive, we need to start with some initial information about resource accesses that are part of the expected attack surface, which we do not have. However, we find that we can easily define which resource accesses should *not* be in P. That is, we can use the *absence* of filters to determine resource accesses that should not be under adversarial control. This complement set of P is $\overline{P}$. We define an approach to calculate $\overline{P}$ below. Any resource access not in $\overline{P}$ is then in P, the expected attack surface.

Formally, a resource access $u \in \overline{P}$ if any of the following conditions are satisfied:

(i) $u \in V_u$: Binding filters do not exist, or
(ii) $u \xrightarrow{e} v \in V_u$: There exists an unfiltered name flow edge originating at u, or
(iii) $(u \xrightarrow{*} v) \wedge (v \in \overline{P})$: There exists a name flow path originating at u to a resource access in $\overline{P}$.

Consider the example in Figure 5. Here, resource accesses a and b filter bindings ($a, b \in V_f$). c does not filter

$a, b, c \in \overline{P}$

Figure 5: Example about determining membership in P

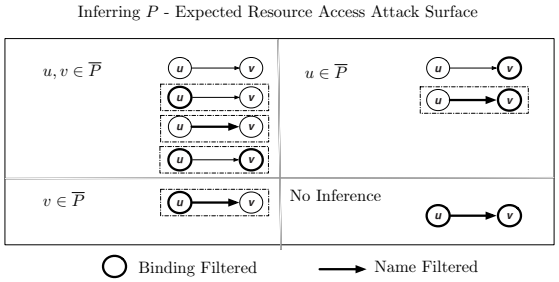


Figure 6: Determining whether a resource access in a resource flow graph should be in $\overline{P}$.

bindings ($c \in V_u$). c's name is determined from input at b, and b's name is determined from input at a. The name flow from a to b is filtered. By *(i)* above, $c \in \overline{P}$ since it does not filter bindings, and the programmer did not expect adversary control by our heuristic. Next, by *(ii)* above, $b \in \overline{P}$, since it is the origin of an unfiltered name flow (which adversaries should not control). Finally, by transitivity using *(iii)* above, $a \in \overline{P}$, because it is the origin of a name flow to a resource access that is in $\overline{P}$, and thus adversaries should not control the name read from resource access at a. All combinations of name and binding filters between a pair of nodes and the inference of node membership in P are presented in Figure 6.

Figure 7 describes the algorithm used to calculate membership in P, given V_f, V_u, E_f, and E_u. It implements *(i)-(iii)* above. It starts by initially assigning any node that does not filter bindings to $\overline{P}$ (*(i)*), and the source of unfiltered name flows to $\overline{P}$ (*(ii)*). It then uses a fixed point iteration to apply the transitive rule *(iii)*, and adds the source of any name flow to a target already in $\overline{P}$ to $\overline{P}$. At the termination of the algorithm, any resource access not in $\overline{P}$ is in P.

5.3 Detecting Missing Filters

Using the name flow graph, we can compute cases where filtering is likely missing. Intuitively, a filter is missing if the program filters *some* adversarial control of resource access but not others. This can happen in two cases: *(a)* if an incoming name flow is filtered but the binding at the resource access is not, or *(b)* a binding is filtered but an outgoing name flow is not. The dotted boxes in Figure 6 show these cases.

Precisely, filters are possibly missing at a resource access r in two cases:

Case 1: $\exists s, e : (s \xrightarrow{e} r \wedge e \in E_f \wedge r \in V_u)$. There exists a filter on an incoming name flow (indicating adversarial control of name) but not a binding filter, or

Input: Set of unfiltered names E_u and bindings V_u
Output: P

```
1:  P ← ∅                          ▷ Resource accesses that can be adversary controlled
2:  for v ∈ V_u do                 ▷ Any node that does not filter bindings
3:      P̄ ← P̄ ∪ v                  ▷ Cannot be adversary controlled
4:  end for
5:  for e ∈ E_u do                 ▷ Any edge that does not filter name
6:      P̄ ← P̄ ∪ e.src              ▷ Mark source as not adversary controlled
7:  end for
8:  c ← True
9:  while c = True do              ▷ Propagate set - fixed point iteration
10:     c ← False
11:     for e ∈ E_u do
12:         if e.tgt ∈ P̄ ∧ e.src ∉ P̄ then
13:             P̄ ← P̄ ∪ e.src
14:             c ← True
15:         end if
16:     end for
17: end while
```

Figure 7: Inferring P from knowledge of filtering

Case 2: $\exists s, e : (r \xrightarrow{e} s \wedge e \in E_u \wedge r \in V_f)$. There exists a filter on a binding (indicating an adversary-accessible resource) but not on all outgoing name flows.

As an example of a missing filter indicating a vulnerability, we found that in the default configuration, the Apache web server filters the name supplied by a client (by stripping `../`), but does not filter the binding used to fetch the HTML file. Therefore, an adversary can create a link of her web page to `/etc/passwd`, which will be served.

Not all possibly missing filters indicate a vulnerability. Some filters perform the same checks as JIGSAW. As an example, we found that `libc` had binding filters when it accessed (some) resources under `/etc` to reject adversary-accessible resources, enforcing Invariant 1 itself. Thus, there is no need to filter names originating from this resource (although *Case 2* indicates a possibly missing filter). We call filters that perform the same checks JIGSAW, redundant.

5.4 Detecting Presence of Binding Filters

We now outline our technique for detecting the filtering of bindings. Our objective in detecting here is to determine resource accesses that perform *any* filtering of bindings. Note that we do *not* aim to prove the correctness of the filtering checks themselves.

To define how we detect binding filters, we discuss how bindings are involved in resource access and how programs filter them. A program accesses many bindings (directories and symbolic links) during a single resource access. In theory, any one of these is controllable by the adversary. Filtering of directories is done by checking its security label, whereas link filtering checks if the binding is a link, and optionally, the security label of the link's

target. Bindings are filtered if, in some cases, the program does not accept a binding based on checks done on any binding used during resource access. An ideal solution would detect the existence of any such check.

Both static and dynamic approaches are possible to detect binding filtering. Static analysis uses the program code to determine if checks exist. However, this is quite challenging as there are a wide variety of ways to perform checks, including, for example, lowering the privilege of the process to that of the adversary [9, 39]. Instead, we opt for a dynamic analysis that detects the effects of filtering.

To detect filters, we have to choose a test that will *definitely fire the filter*, if such a filter is present. Our tests are attacks that attempt to exploit vulnerabilities if filters were absent. Not all attacks corresponding to vulnerability classes in Table 1 are suitable as tests to detect program filters. Consider the subset of attacks corresponding to vulnerability classes in Table 1 where the adversary uses her control of bindings to direct the victim to an adversary-accessible resource (Row 1). If the program accepts the adversary-accessible resource, it is generally not possible to determine if this was due to the program intentionally accepting this resource or due to the program assuming that there would be no adversary control of the resource access. On the other hand, consider the subset of attacks corresponding to vulnerabilities where the adversary uses control of bindings to direct the victim to an adversary-inaccessible resource (e.g., link following). Here, if the programmer were expecting adversary-controlled bindings, she *has* to add checks to block this resource access as this scenario is, by definition, a confused deputy vulnerability. Thus, we can use the results of a link following vulnerability to determine the existence of binding filters, and thus, the programmer's expectation. In Section 8, we describe a dynamic analysis framework that performs these tests.

5.5 Detecting Presence of Name Filtering

The other way for adversaries to control resource access is to control names. We aim to determine if the program makes any attempt to filter names, which would indicate that the programmer expected to receive an adversary-controlled name. Again, note that to determine programmer expectation, we only need to determine if there is *any* filtering at all, not if the filtering is correct.

To determine name filters in programs, we first describe how names originate. Names are either hard-coded in the program or received at runtime. First, hard-coded names are constants defined in the program binary or a dynamic library. For an adversary to have control of hard-coded names, she needs to control the binary or library, in which case trivial code attacks are possible. Therefore, we assume hard-coded names to not be under adversarial control. Second, programs get names from runtime input. In Figure 1, a client is requesting a HTML file by supplying its name. The server reads the name from this request (name source) and accesses the HTML file resource from this client input (name sink). In general, a name can be computed from input using one or more read system calls.

Next, we define the action of filtering names. There are two ways in which programs filter names. First, programs can directly manipulate the name. For example, web servers strip malicious characters (e.g., `..`) from names. Second, it can check that the resource retrieved from this name is indeed accessible to the adversary. For example, the setuid `mount` program accepts a directory to mount from untrusted users who are potential adversaries, but checks that the user indeed has write permissions to the directory before mounting. Thus, a name is filtered between a source and a sink if, in some cases, the name read at a source is changed, or the resource access at the sink is blocked. An ideal solution would detect the existence of any such checks.

Determining name filtering is a two-step process. First, we should determine pairs of resource accesses where the name is read from one resource (source) and used in the other (sink). Next, we determine if the program places any filters between this source-sink pair.

Again, we can use static or dynamic analysis to find pairs and filters. To detect filters, Balzarotti et al. used string analysis [4], whereas techniques such as weakest preconditions [7] or symbolic execution [18] can also be used. However, static analysis techniques are traditionally sound, but may produce false positives. Therefore, we use dynamic analysis to detect evidence of filtering.

To determine both pairs and filtering, we use a runtime analysis inspired by Sekar [33]. Sekar's aim is to detect injection attacks in a black-box manner. The technique is to log all reads and writes by programs, and find a correlation between reads and writes using an approximate string matching algorithm. Thus, given as input a log of the program's read and write buffers, the algorithm returns true if a write buffer matches a read buffer "approximately".

We adapt this technique to find name flows. We log all reads and names during resource access, and find matches between names and read buffers. We first try matching the full name; if no match is found, we try to match the directory path and final resource separately. Often, parts of a name are read from data of different resources. For example, a web server's document root is read from the configuration file, whereas the file to be served is read from the client's input. Both of these are combined to form the name of the resource served. As with the method for finding binding filters, we use the directory traversal attack in Row 2 to trigger filtering.

Since our analysis is a black-box approach, if a possible name flow is found, the read buffer might just coincidentally happen to have the name, but not actually flow to it. Thus, we execute a verification step. We run the test suite again, but this time change the read buffer containing the name to special characters, noting if the name also changes. If it does, we have found a name flow.

6 Phase 2: Enforce Programmer Expectations

Once we find the expected attack surface P, JIGSAW enforces resource access protections using Invariant 1 and Invariant 2 in Section 3 on program deployments. To do this, a reference monitor [1] has to mediate all resource accesses and enforce these rules. To enforce these rules correctly for each resource access, a reference monitor must determine whether this resource access is in P, and identify the system deployment's attack surface S and adversary accessibility to resources R.

6.1 Protecting Accesses in P at Runtime

The first challenge is to determine whether the resource access is in P. There are two ways to do this: (a) the program code can be modified to convey its expectation to the monitor through APIs, or (b) the monitor already knows the program expectation and identifies each resource access. Capability systems use code to convey their expectation during each resource access. Capability systems [21] present capabilities for only the expected resources to the OS during access. For example, decentralized information flow control (DIFC) systems [19,45] require the programmer to choose labels for the authorized resource for each resource access. However, such systems require modifying program code and recompilation, which can be complex to do correctly.

Another option is for the reference monitor to extract information necessary for it to identify the specific resource access, and hence whether it is in P. Researchers have recently made the observation that if they only protect a process, they may introspect into the process (safely) to make protection decisions [40]. They implemented a mechanism called the Process Firewall, a Linux kernel module that introspects into the process to enforce rules to block vulnerabilities per system call invocation. This is similar in concept to a network firewall that protects a host by restricting access per individual firewall rules. We use this option because it does not require program code or system access control policy changes, and was shown to be much faster than corresponding program checks in some cases.

The general invariant that the Process Firewall enforces is as follows:

`pf_invariant(subject,` **entrypoint**`,`**syscall_trace**`,`
`object,` **resource_id**`,`**adversary_access**`, op)` $\mapsto$ `Y|N`

Here, **entrypoint** is the user stack at the time of the system call. Resource accesses in P are identified by their entrypoint. A single system call may access multiple bindings (e.g., directories and links) and a resource. As each binding and resource is accessed at runtime, its adversary access is used in the decision. If a binding is adversary-accessible, then the resource access is in S. If the final resource is adversary-accessible, then the resource access is in R. If a resource access in R is the source of a name, this fact is recorded in **syscall_trace** and the resource access using this name is in S. This general invariant is instantiated to enforce our invariants in Section 3. The invariants are converted into Process Firewall rules using templates (Section 8).

6.2 Finding Adversary Accessibility R

R is the set of resource accesses at runtime that use adversary accessible resources, and is required to enforce Invariant 2 in Section 3. Calculating R requires knowing: *(a)* who an adversary is, and *(b)* whether the adversary has permissions to access resources. We address these questions in turn.

There have been several heuristics to determine who an adversary is. Gokyo [17] uses the system's mandatory access control policy to determine the set of SELinux labels that are trusted for the system – the rest are adversarial. Vijayakumar *et al.* [41] extend this approach to identify per-program adversaries. Chari et al. [10] and Pu et al. [43] use a model based on discretionary access control – a process running with a particular user ID (UID) has as its adversaries any other UID, except for the superuser (`root`). We can use any of these approaches to define an adversary.

Second, we need to determine whether an adversary has permissions to resources. As discussed in Section 2, an adversary-accessible resource is one that the system's access control policy Y allows an adversary permissions to (read for secrecy vulnerabilities, write for integrity vulnerabilities, and execute for both). This can be queried directly from the access control policy Y. Any resource access at runtime that uses adversary-accessible resources is in R.

Some resources become adversary-accessible through indirect means. For example, programs log adversarial requests to log files. Thus, adversaries affect data in log files even if the access control policy does not give adversaries direct write permissions to log files. Such exceptional resources are currently manually added to R.

6.3 Finding Deployment Attack Surface S

The deployment attack surface S is the set of resource accesses a process performs at runtime that are adversary-controlled. An adversary can control resource accesses

by controlling either the name or a binding (or both). An adversary controls a binding if she uses her write permissions in a directory to create a binding. An adversary controls a name if the adversary has write permission to the resource the name is fetched from.

To determine adversary control of names, we need to detect if there is a data flow from adversary-supplied data to a name used in a resource access. Data flow is most often determined by taint tracking [5, 20, 22, 31]. However, taint tracking techniques have overheads ranging from 2× to 50× [28]. Instead, JIGSAW approximates data flow using control flow (**entrypoint**s – process stacks at the time of reading names and using names). Pairs of process stacks during read and resource access system calls are initially associated by detecting explicit data flow between these calls (Section 5.5). During enforcement, if the Process Firewall sees the same stacks, we assume an explicit data flow between the read and resource access system calls, and the resource access is in S.

6.4 Finding Vulnerabilities

We can use the rules generated to also find vulnerabilities. Vulnerabilities are detected whenever a resource access is denied by our rules but is allowed by the program.

We use the same dynamic analysis from test suites that we use to detect the presence of filters in Sections 5.4 and 5.5 to also test the program for vulnerabilities in our particular deployment. Instead of enforcing the rules, we compare denials by Invariant 1 or Invariant 2 in Section 3 with whether the program allows the resource access. If the rule denies resource access whereas the program accepts the resource, we flag a vulnerability. Note that this process locates vulnerabilities in our specific deployment; there might be other vulnerabilities in other deployments that we miss. In any case, our rules, if enforced, will protect these program vulnerabilities in any deployment.

7 Proving Security of Resource Access

In this section, we first argue that if P, S and R are calculated perfectly, then JIGSAW eliminates resource access vulnerabilities. Our argument is oracle-based; that is, we can reason about the correctness of our approach assuming the correctness of certain oracles on which it depends. Our argument is contingent on the correctness of the three oracles that determine: *(i)* program expectation for P, *(ii)* adversary accessibility of resources for R, and *(iii)* adversary control of names and bindings for S. We then discuss the practical limitations JIGSAW faces in realizing these oracles.

7.1 Theoretical Argument

Assuming ideally correct oracles for determining programmer expectation, adversary accessibility of resources and adversary control of names and bindings, we argue that Invariants 1 and 2 in Section 3 protect a program from all resource access vulnerabilities as defined in Section 2.1 without false positives.

According to the definition in Section 2.1, a resource access vulnerability is caused when an adversary controls an input (name or binding) to direct a program to an adversary-accessible resource when the program expects an adversary-inaccessible resource (and vice-versa). Our proof hinges on two observations. First, resource access vulnerabilities are impossible if adversaries do not control the input name or binding. Invariant 1 denies all adversary control of inputs where the programmer expects only adversary-inaccessible resources, thus eliminating all vulnerabilities in these cases. Thus, vulnerabilities are only possible where the programmer expects to adversary control of input name or binding. Second, if indeed the adversary controls the input name or binding, the only authorized output is an adversary-accessible resource; otherwise, a confused deputy vulnerability (Row 2 in Table 1) can result. To block this, Invariant 2 allows retrieval of only adversary-accessible resources when input is under adversary control. Hence, we have shown that our rules deny resource accesses *if and only if* adversary control of input directs the program to unexpected resources, thus blocking resource access vulnerabilities without false positives.

7.2 Practical Limitations

In a practical setting, the determination of program expectations, adversary accessibility to resources, and adversary control of names and bindings is imperfect. This may lead to false positives and false negatives. We will discuss limitations with determining each of these in turn.

The first oracle determines programmer expectation, for which we use the intuitive heuristic in Section 5: if a programmer does not place filters, then she does not expect adversary control of resource access, i.e., she only expects adversary-inaccessible resources. The detection of filters themselves uses runtime analysis. This faces three issues: *(i)* if identified filters are actually present, *(ii)* if actual filters are missed, and *(iii)* incompleteness of runtime analysis. First, if a detected filter is not actually present, this might result in false negatives. However, to detect filters, we mimic an attack and detect if the program blocks the attack. The only way the program could have blocked the attack is if it had a filter. Second, if an actual filter is missed, this might result in

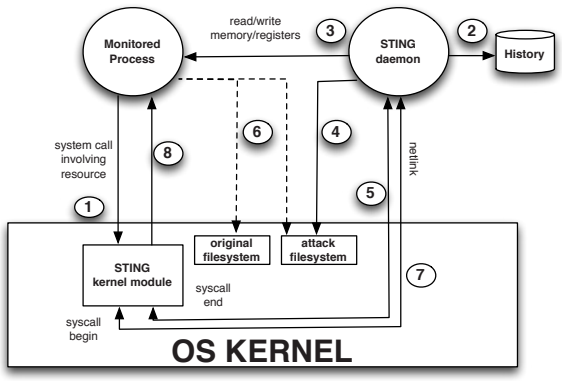


Figure 8: Implementation of JIGSAW's testing framework.

false positives. By the same argument above, if the filter cannot defend against our mimicked attack, then it is not complete enough anyway. Third, runtime analysis is inherently incomplete and may lead to false negatives for those resource accesses not covered. However, even with the limited developer test suites currently available, we were able to generate rules to block many vulnerabilities and find previously-unknown ones even in mature programs.

Next, we have the oracle that determines adversary accessibility to resources. The main challenge here is determining who an adversary is. If we do not have a sufficiently strong adversary model, we may miss adversaries and hence have false negatives. While there is no universally agreed-upon definition of who an adversary is, we use the intuitive DAC model[5] that most programmers assume [10, 43]. However, our framework permits the use of different adversary models. More conservative adversary models [41] will identify more adversary-accessible resources, possibly exposing more vulnerabilities.

The final oracle determines adversary control of names and bindings. The challenge here is to determine if there is a data flow from adversary-supplied data to a name used in a resource access. As described in Section 6.3, JIGSAW approximates data flow using control flow. However, even if control flow is the same across two executions of the program, it does not necessarily imply the data flow is the same, leading to false positives and negatives. While we have not found this to be a problem in our experiments (Section 9.4), more precise data flow tracking techniques [5, 20, 31] will address this challenge.

8 Implementation

There are two parts to our implementation. First, we need to test individual program resource accesses to detect the presence of filters. This is used by the algorithm in Figure 7 to generate P. Second, we need to enforce invariants in Section 3 using the Process Firewall. This involves determining R and S; this is done as discussed in Section 6.2 and Section 6.3 respectively.

8.1 Testing Programs

To test programs, we develop a framework that can intercept system calls and pass control to a user-space component that performs the tests. The kernel component is a small module that intercepts system calls and returns, and forwards them to a user-space daemon through netlink sockets. The flow of operations is as shown in Figure 8. When a monitored program makes a system call, it is intercepted by the framework's kernel module, and forwarded to the user-space daemon. There are two resource namespaces available per program – a "test" namespace that is modified for hypothetical tests and the original namespace (similar in principle to [39]). This daemon introspects into the monitored process to identify its resource access (using the user-space stack), and checks its history to see if filters have already been detected. If not, it then proceeds to modify the test filesystem (for binding filter detection). It then returns to the kernel module. Control passes to the process in kernel mode, which accesses the original or test filesystem (depending on whether binding filters are being tested). The system call end is also intercepted, and similarly forwarded to the user-space daemon to test for name filters (as the read buffer is now available and can be modified).

We use test suites provided with program source code to drive the programs. We repeatedly run these suites until all resource accesses have been tested for filters.

8.2 Enforcing Invariants

As noted, JIGSAW uses the open-source Process Firewall [40] to perform enforcement. The Process Firewall is a kernel module that uses the Linux Security Modules to mediate resource accesses. In addition, it can perform a user stack backtrace to identify the particular resource access being made. Given P and the edges in the name flow graph, we have two rule templates to instantiate invariants into rules to be enforced by the Process Firewall. Figure 9 shows the templates. Note that the rules for confused deputy are stateful. Adversary control of name or binding is recorded by the first rule, the adversary's identity is recorded by the second rule, and the third rule uses

[5] A process with uid X has as its adversaries any uid $Y \neq X$ (except superuser `root`)

Rule Templates

Unexpected Adversary Control: $(r \in S) \wedge (r \in \overline{P}) \implies Deny$
For each $r \in \overline{P}$:
```
pftables -i r.ept -d LOW -o DIR_SEARCH -j DROP
```

Confused Deputy: $(r \in P) \wedge (r \in S) \wedge (r \in \overline{R}) \implies Deny$
Name:
For each $r_1 \in P$ such that $E(r_1, r_2)$:
```
pftables -i r1.ept -d LOW -j STATE --set --key
<random_value> --value 1
pftables -i r1.ept -d LOW -j STATE --set --key "adv"
--value ADV_ID
pftables -i r2.ept -d HIGH -m STATE --key
<random_value> --cmp 1 --equal -m PERM -s --key
"adv" --deny -j DROP
```
Binding:
```
pftables -d LOW -o DIR_SEARCH -j STATE --set --key
<r1.ept> --value 1
pftables -d LOW -o DIR_SEARCH -j STATE --set --key
"adv" --value ADV_ID
pftables -d HIGH -m STATE --key <random_value> --cmp
1 --equal -m PERM -s --key "adv" --deny -j DROP
```

Figure 9: Process Firewall rule templates.

this state to block access to an adversary-inaccessible resource. Thus, R is adversary-specific; in addition to protecting programs against all its adversaries, it also prevents one adversary from using the program as a confused deputy against another adversary.

9 Evaluation

In this section, we evaluate our technique on several widely-used programs. We chose these programs because: *(i)* resource accesses are central to their operation, and *(ii)* they offer a study in contrast – OpenSSH and Postfix were explicitly architected for security [30], whereas the others were not. To derive expectations, we used developer test suites that came with the program or created our own. We answer: (a) how common are implicit programmer expectations during resource access, (b) whether the resulting expected attack surface was safe for our deployment and vulnerabilities where not, and (c) security effectiveness of hardened programs from resource access vulnerabilities. We find that in all programs except OpenSSH, more than 55% of all resource accesses are implicitly expected to be free from adversarial control. Moreover, we discovered two previously-unknown vulnerabilities and one default misconfiguration in the Apache webserver. Finally, we find that protection can be enforced with an overhead of $<$6% on a variety of programs and few false positives.

9.1 Implicit Programmer Expectations

Table 2 shows a summary of the results obtained by JIGSAW. We first note the percentage of resource accesses

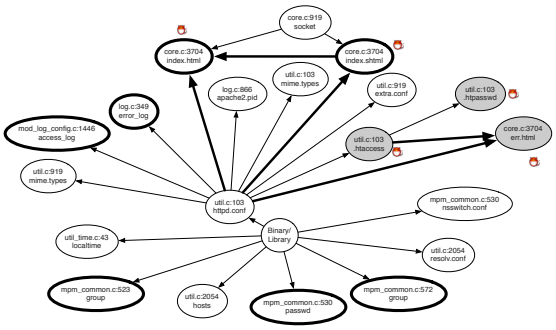


Figure 10: Resource Flow Graph for Apache. Nodes that have the icon of adversaries beside them are those where we found adversarial control of resource access in our deployment.

that are implicitly expected to *not* be in P, due to the absence of name or binding filters. For all programs except OpenSSH, the programmer placed no filters for more than 55% of resource accesses. If any of these resource accesses somehow come under adversarial control, the program can be compromised. It is very easy for OS distributor policies or administrator configurations to not match with these programmer assumptions. By explicitly identifying such resource accesses, we are able to protect them from any adversarial access in *any* deployment. OpenSSH makes fewer assumptions during resource access (17.6%). However, OpenSSH was re-architected after several years of experience with previous vulnerabilities. Using our technique, we can protect all resource accesses.

9.2 Case Study: Apache

In total, we found 20 resource accesses for Apache. Of these, Apache code filtered bindings for 7 accesses, and the name for 5 accesses. This led to 13 out of 20 resource accesses (65%) *not* being in P (using the algorithm in Figure 7). We found three resource accesses in $S-P$ for the Apache web server in our deployment, violating the first rule in Section 3. These corresponded to two previously unknown vulnerabilities in the Apache web server and one default misconfiguration. That such problems occur in even a mature program like Apache shows the importance of principled reasoning of resource access. While we found these vulnerabilities in our deployment, other deployments may have different vulnerabilities, but all will be blocked using our enforcement (Section 9.3).

Figure 10 shows the resource flow graph for Apache. Apache's expected attack surface is centered around resource accesses during the interaction with a client to serve a web page. It assumes that the location of the main configuration file and resources specified in it are not adversary controlled. Apache's resource flow graph is relatively complex due to long chains of resource flows, and it is difficult to reason about safety without the help of automated tools like ours. Resource accesses that had vul-

Program	*Dev Tests?*	$\|V\|$	$\|E\|$	$\|V_f\|$	$\|E_f\|$	$\in P$	$\notin P$	*Impl. Exp. %*	*Missing*	*Redundant*	*Vulns.*	*Inv. 1s*	*Inv. 2s*
Apache v2.2.22	Yes*	20	23	7	5	7	13	65%	2	0	3	13	12
OpenSSH v5.3p1	Yes	17	17	14	0	14	3	17.6%	0	3	0	3	2
Samba3 v3.4.7	Yes	210	84	78	19	78	132	62.8%	0	5	0	132	40
Winbind v3.4.7	Yes	50	38	19	13	19	31	63.3%	0	0	0	31	28
Postfix v2.10.0	No	181	15	79	7	79	102	56.32%	0	9	0	102	15

Table 2: Statistics of program-wide resource accesses. *Dev Tests* show whether we used developer test suites or created our own. *Impl. Exp.* is the percentage of resource accesses ($|\overline{P}|/|V|$) that are implicitly expected to be adversary-inaccessible. The last two columns show the number of instantiations of Invariant 1 and Invariant 2 in Section 3 for resource accesses in the program. *- We augmented the Apache test suite with additional tests.

nerabilities in our deployment are shaded in the graph.

The first vulnerability we found was during resource access of a user-defined `.htpasswd` file. Apache allows each user the option of enabling HTTP authentication for parts of their website. This includes the ability to specify a password file of their choice. However, the resource access that fetches this password file is not filtered. Thus, users can specify any password file – even one that they do not have access to. One example exploit is to direct this password file to be the system-wide `/etc/passwd`. Traditionally, it is difficult to brute-force the system-wide password file since prompts are rate-limited. However, since HTTP authentication is not rate-limited, this may make such brute-force attacks realistic. Such a scenario, though obvious after discovery, is very difficult to reason about manually due to Apache's complex resource accesses. Thus, it has remained hidden all these years.

The second vulnerability is a default misconfiguration. When serving web pages, Apache controls whether symbolic links can be followed from user web pages by the option `FollowSymLinks`, which is turned on by default in Ubuntu and Fedora packages. Turning this option on implicitly assumes trust in the user to link to only her own web pages. Interestingly, the name for this resource access is filtered – only the bindings are not. One way we were able to take advantage of this misconfiguration was through the error document on specific errors, such as HTTP 404, that is specifiable in the user-defined configuration `.htaccess` file. This allows an adversary to access any resource the Apache web server itself can read, for example, the password file and SSL private keys. We found that our department web server also had this option turned on. By simply making an error document linked to `/etc/passwd`, we were able to view the contents of the password file on the server. This demonstrates another typical cause of resource access attacks – administrators misconfiguring the program and violating safety of the expected attack surface.

The third vulnerability is a link following attack on `.htaccess`. Apache allows `.htaccess` to be any file on the filesystem it has access to. This may be exploited to leak configuration information about the webserver.

Finally, we note that test suites that come with programs are traditionally focussed towards testing functionality and not necessarily resource access. For example, the stock test suite for Apache only uncovered 7 resource accesses in total, and after we augmented it, there were 20 in total. Better test suites for resource access would help test more resource accesses.

9.3 Process Firewall Enforcement

Process Firewall rules enforce the safety of the expected attack surface under the deployment attack surface. Given the program's expected attack surface, Process Firewall rules enforce that any adversary-controlled resource access at runtime (i.e., part of the deployment attack surface) is allowed only if the resource access is also part of the program's expected attack surface. In addition, for those resource accesses allowed, they also protect the program against confused-deputy link and directory traversal vulnerabilities. The last two columns in Table 2 shows the number of Process Firewall rules we obtained (separately due to Invariants 1 and 2).

We evaluated the ability of rules to block vulnerabilities. First, we verified the ability of these rules to block the three discovered vulnerabilities in Apache. Second, we tried previously-known, representative resource access vulnerabilities against Apache and Samba. We tested an untrusted library load (CVE-2006-1564) against Apache. Here, a bug in the package manager forced Apache to search for modules in untrusted directories. Our tool deduced that the resource access that loaded libraries did not have any filtering, and thus, was not in P, blocking this vulnerability due to Invariant 1 in Section 3. In addition, we tested a directory traversal vulnerability in Samba (CVE-2010-0926). This is a confused deputy vulnerability involving a sequence of `../` in a symbolic link. This vulnerability was blocked due to Invariant 2.

9.4 False Positives

As discussed in Section 7.2, false positives are caused by improper determination of: *(i)* programmer expectation and *(ii)* adversary control of names.

First, false positives are caused by a failure of our heuristic in Section 3 that determines program expectation. In some cases, we found that a program had no filters at a resource access, but still expected adversary-controlled resource access. We found that this case occurs in certain small "helper" programs that perform a requested task on a resource without any resource access filters. For example, consider that the administrator (`root`) runs the `cat` utility on a file in an adversarial user's home directory. Because `cat` does not filter the input bindings, the user can always launch a link following attack by linking the file to the password file, for example. However, if there is no attempted attack, then our rule will block `cat` from accessing the user's file, because the resource access has no filters and is thus not part of the expected attack surface (by our heuristic). However, we may want to allow such access, because `cat` has filters to protect itself from the input data to prevent vulnerabilities such as buffer overflows.

To address such false positives, we propose enforcing protection for such helper programs specially. Our intuition is that when these programs perform adversary-controlled resource access, they can be considered adversarial themselves. All subsequent resources to which data is output by these programs are then considered adversary-accessible. Other programs reading these resources should protect themselves from input (e.g., names) as if they were dealing with an adversary-accessible resource.

To enforce this approach, we implemented two changes. First, we enforce only Invariant 2 (confused deputy) in Section 3 for these programs. Second, whenever Invariant 1 would have disallowed access, we instead allow access, but "taint" all subsequent output resources by marking them with the adversary's label (using filesystem extended attributes).

We evaluated the effect of this approach during the bootup sequence of our Ubuntu 10.04 LTS system. We manually identified 15 helper programs. During boot, various startup scripts invoked these helper programs a total of 36 times. In our deployment, 9 of these invocations accessed an adversary-accessible resource. Note that our original approach would have blocked these 9 resource accesses, disrupting the boot sequence, whereas our modification allows these resource accesses. These invocations subsequently tainted 4 output resources – two log files and two files storing runtime state. We found two programs reading from these tainted files – `ufw` (a simplified firewall), and the `wpa_supplicant` daemon (used to manage wireless connections). These programs will find the tainted resources adversary-accessible, and will have to protect themselves from such input.

Second, false positives are caused during enforcement by our implementation's approximation of adversarial data flow using control flow. Such false positives are due to implementation limitations and not any fundamental shortcoming of our approach. To evaluate this, we used two separate test suites for Apache – one to build the name flow graph and generate Process Firewall rules, and the other to test the produced rules. We used our enhanced test suite to generate rules and ApacheBench to test the generated rules. ApacheBench ran without any false positives. However, different configurations or variable values might result in different data flows even if the stacks remain the same. As mentioned in Section 7.2, accurate data flow tracking can solve this problem.

9.5 Performance

A detailed study of the Process Firewall is in [40]. In summary, system call microbenchmarks showed overheads of up to 10.6%, whereas macrobenchmarks had overheads of up to 4%. The main cause for overhead is unrolling the process stack to identify the system call. To confirm these results, we evaluated the performance overhead of a hardened Apache webserver (v2.2.22) that had the 25 rules from Table 2. Using ApacheBench to request the default Apache static webpage, we found an overhead of 4.33% and 5.28% for 1 and 100 concurrent clients respectively. However, we can compensate for such overhead by compiling Apache without resource access filters, since filters are now redundant given our enforced rules. Vijayakumar *et al.* [40] showed that removing code filters causes a throughput increase of up to 8% in Apache.

10 Related Work

10.1 Inferring Expectations

Determining programmer expectations from code has previously been done in a variety of contexts. Engler [14] et al. infer programmer *beliefs* from code. For example, if a pointer `p` is dereferenced, the inferred belief is that `p != NULL`. They use this to find bugs in the Linux kernel. Closely related are techniques to infer invariants from dynamic traces [3, 15, 32]. Daikon [15] uses dynamic traces to infer hypothesis such as that a particular variable is less than another. Baliga *et al.* [3] use Linux kernel traces to propose invariants on data structures. While we deal with a different class of attacks, our approach is in the same spirit as the above works.

10.2 Defenses During Resource Access

Current defenses against resource access attacks in OSes are ad-hoc and fundamentally limited. Defenses can be classified into those that require changes to either the *(i)* system (e.g., OS, libraries), or *(ii)* program code.

The simplest system defense is to change the access control policy to allow a process access to only the set of expected resources. Unfortunately, this defense is both complicated and does not entirely stop resource access attacks. First, fixing access control policies is a complicated task. For example, even the minimal (targeted) SELinux MAC policy on the widely-used Fedora Linux distribution has 95,600 rules. Understanding and changing such rules requires domain specific expertise that not all administrators have. Second, access control alone is insufficient to stop resource access attacks because it treats processes as a black-box and does not differentiate between different resource access system calls. In our example in Figure 1, the web server opens both a log file and user HTML pages. Thus, it needs permissions to both resources. However, it should not access the log file when it is serving a user HTML page, and vice versa. Traditional access control does not make this difference. Other system defenses have mainly focused on TOCTTOU attacks [11, 13, 25–27, 35–38, 44] and link following [10]. However, system defenses are prone to cause false positives because they do not know what programs expect [8], e.g., which pairs of system calls expect to access the same resource.

The simplest program defense is to use program code filters that accept only the expected resources. However, there are a number of challenges to writing correct code checks. First, such checks are *inefficient* and cause performance overhead. For example, the Apache web server documentation [2] recommends switching off resource access checks during web page file retrieval to improve performance. Second, checks are *complicated*. The system-call API that programs use for resource access is not atomic, leading to TOCTTOU races. There is no known race-free method to perform an `access-open` check in the current system call API [8]. Chari *et al.* [10] show that to defend link following attacks, programmers must perform at least four additional system calls per path component for each resource access. Going back to the example in Figure 1, the checks on lines 7 and 8 are not enough – the correct sequence to use is `lstat-open-fstat-lstat` [10]. Thirdly, program checks are *incomplete*, because adversary accessibility to resources is not sufficiently exposed to programs by the system-call API. Currently, programs can query adversary accessibility only for UNIX discretionary access control (DAC) policies (e.g., using the `access` system call), but many UNIX systems now also enforce mandatory access control (MAC) policies (e.g., SELinux [24] and AppArmor [23]) that allow different adversary accessibility. While custom APIs have been proposed [19, 29, 34, 42] to address such limitations, these require additional programmer effort and do not protect current programs.

11 Conclusion

In this paper, we presented JIGSAW, an automated approach to protect programs from resource access attacks. We first precisely defined resource access attacks, and then noted the fundamental cause for them – a mismatch between programmer expectations and the actual deployment the program runs in. We defined two invariants that, if evaluated and enforced correctly, can theoretically offer complete protection against resource access attacks. We proposed a novel technique to evaluate programmer expectations based on the presence of filters, and showed how the invariants could practically be enforced by the Process Firewall.

We applied this technique to harden widely-used programs, and discovered that programmers make a lot of implicit assumptions. In this process, we discovered two previously-unknown exploitable vulnerabilities as well as a default misconfiguration in Apache, the world's most widely used web server. This shows that even mature programs only reason about resource access in an ad-hoc manner. The analysis as presented in this paper can thus efficiently and automatically protect programs against resource attacks at runtime.

Acknowledgements

We thank the anonymous reviewers and our shepherd David Evans for their insightful comments that helped improve the presentation of the paper. Authors from Penn State acknowledge support from the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-FA9550-12-1-0166. Mathias Payer was supported through the DARPA award HR0011-12-2-005. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.

[2] Apache Performance Tuning. `http://httpd.apache.org/docs/2.2/misc/perf-tuning.html#symlinks`, 2012.

[3] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference*, pages 77–86, Anaheim, California, USA, December 2008. IEEE Computer Society Press, Los Alamitos, California, USA.

[4] D. Balzarotti *et al.* Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[5] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proceedings of POLICY'02*. IEEE Computer Society, 2002.

[6] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.

[7] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, 2007.

[8] X. Cai *et al.* . Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE SSP '09*, 2009.

[9] S. Chari and P. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transaction on Infomation and System Security*, 6:173–200, May 2003.

[10] S. Chari *et al.* Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS '10*, 2010.

[11] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[12] Common vulnerabilities and exposures. `http://cve.mitre.org/`.

[13] D. Dean and A. Hu. Fixing races for fun and profit. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[16] N. Hardy. The confused deputy. *Operating Systems Review*, 22(4):36–38, Oct. 1988.

[17] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[19] M. N. Krohn *et al*. Information flow control for standard OS abstractions. In *SOSP '07*, 2007.

[20] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of ACSAC '06*, pages 463–472. IEEE Computer Society, 2006.

[21] H. M. Levy. *Capability-based Computer Systems*. Digital Press, 1984. Available at `http://www.cs.washington.edu/homes/levy/capabook/`.

[22] J. Newsome *et al* . Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[23] Novell. AppArmor Linux Application Security. `http://www.novell.com/linux/security/apparmor/`.

[24] Selinux. `http://www.nsa.gov/selinux`.

[25] OpenWall Project - Information security software for open environments. `http://www.openwall.com/`, 2008.

[26] J. Park, G. Lee, S. Lee, and D.-K. Kim. Rps: An extension of reference monitor to prevent race-attacks. In *PCM (1) 04*, 2004.

[27] M. Payer and T. R. Gross. Protecting applications against tocttou races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2012.

[28] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 115–126, Berkeley, CA, USA, 2013. USENIX Association.

[29] D. E. Porter *et al.* Operating system transactions. In *SOSP '09*, 2009.

[30] N. Provos *et al* . Preventing privilege escalation. In *USENIX Security '03*, 2003.

[31] F. Qin *et al* . LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO 39*, 2006.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[33] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[34] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, December 1999.

[35] K. suk Lhee and S. J. Chapin. Detection of file-based race conditions. *Int. J. Inf. Sec.*, 2005.

[36] D. Tsafrir *et al.* Portably solving file tocttou races with hardness amplification. In *USENIX FAST*, 2008.

[37] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, pages 243–255, 2003.

[38] P. Uppuluri, U. Joshi, and A. Ray. Preventing race condition attacks on file-systems. In *SAC-05*, 2005.

[39] H. Vijayakumar, J. Schiffman, and T. Jaeger. Sting: Finding name resolution vulnerabilities in programs. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, August 2012.

[40] H. Vijayakumar, J. Schiffman, and T. Jaeger. Process firewalls: Protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems (EUROSYS 2013)*, April 2013.

[41] H. Vijayakumar *et al.* Integrity walls: Finding attack surfaces from mandatory access control policies. In *ASIACCS*, 2012.

[42] R. Watson *et al.* Capsicum: practical capabilities for UNIX. In *USENIX Security*, 2010.

[43] J. Wei *et al.* Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *USENIX FAST '05*, 2005.

[44] J. Wei *et al.* A methodical defense against TOCTTOU attacks: the EDGI approach. In *IEEE International Symp. on Secure Software Engineering (ISSSE)* , 2006.

[45] N. Zeldovich *et al* . Making information flow explicit in HiStar. In *OSDI '06*, 2006.

Static Detection of Second-Order Vulnerabilities in Web Applications

Johannes Dahse
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
johannes.dahse@rub.de

Thorsten Holz
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
thorsten.holz@rub.de

Abstract

Web applications evolved in the last decades from simple scripts to multi-functional applications. Such complex web applications are prone to different types of security vulnerabilities that lead to data leakage or a compromise of the underlying web server. So called *second-order vulnerabilities* occur when an attack payload is first stored by the application on the web server and then later on used in a security-critical operation.

In this paper, we introduce the first automated static code analysis approach to detect second-order vulnerabilities and related multi-step exploits in web applications. By analyzing reads and writes to memory locations of the web server, we are able to identify unsanitized data flows by connecting input and output points of data in *persistent data stores* such as databases or session data. As a result, we identified 159 second-order vulnerabilities in six popular web applications such as the conference management systems *HotCRP* and *OpenConf*. Moreover, the analysis of web applications evaluated in related work revealed that we are able to detect several critical vulnerabilities previously missed.

1 Introduction

Web applications are the driving force behind the modern Web since they enable all the services with which users interact. Often, such applications handle large amounts of (potentially sensitive) data such as text messages, information about users, or login credentials that need to be stored persistently on the underlying web server. Further, *sessions* are used to temporarily store data about a user interacting with the web application during multi-step processes. All of this data can potentially be abused by an attacker to cause harm. Many different kinds of attacks against web applications such as *Cross-Site Scripting* (XSS) or *SQL injection* (SQLi) attacks are known and common injection flaws are well understood. Such attacks can be prevented by sanitizing user input and many approaches to address this problem were presented in the last few years (e.g., [2, 8, 15, 21, 22, 24, 27, 29]).

One common assumption underlying many detection and prevention approaches is that data that is already stored on the server is safe. However, an adversary might be able to bypass the defenses via so called *second-order vulnerabilities* if she manages to first abuse the web application to store the attack payload on the web server, and then later on use this payload in a security-critical operation. Such vulnerabilities are often overlooked, but they can have a severe impact in practice. For example, XSS attacks that target the application's users are worse if the payload is stored in a shared resource and distributed to all users. Furthermore, within *multi-step exploits* a vulnerability can be escalated to a more severe vulnerability. Thus, detecting second-order vulnerabilities is crucial to improve the security of web applications.

Detecting Second-Order Vulnerabilities To prevent such attacks, the source code of a given web application is assessed before it is deployed on a web server. This can be done either via dynamic or static analysis. There are several dynamic approaches to detect second-order XSS attacks via fuzzing [14, 19]. Generally speaking, such approaches try to inject random strings to all possible POST request parameters in a black-box approach. In a second step, the analysis tools determine if the random string is printed by the application again without another submission, indicating that it was stored on the web server. However, the detection accuracy for second-order vulnerabilities is either unsatisfying or such vulnerabilities are missed completely [4, 7, 13, 23]. Artzi et. al. [1] presented a dynamic code analysis tool that considers persistent data in sessions, but their approach misses other frequently used data stores such as databases or files. Furthermore, one general drawback of dynamic approaches is the typically low code coverage.

Static code analysis is a commonly used technique to find security weaknesses in source code. Taint analysis and similar code analysis techniques are used to study the data flow of untrusted (also called *tainted*) data into critical operations of the application. However, web applications can also store untrusted data to external resources and later on access and reuse it, a problem that is over-

looked in existing approaches. Since the data flow is deferred and can be split among different files and functions of the application, second-order vulnerabilities are difficult to detect when analyzing the source code statically. Furthermore, static code analysis has no access to the external resources used by the application and does not know the data that is stored in these.

We are not aware of any plain *static* code analysis implementation handling second-order vulnerabilities. The main problem is to decide whether data fetched from persistent stores is tainted or not. Assuming all data to be tainted would lead to a high number of false positives, while a conservative analysis might miss vulnerabilities.

Our Approach In this paper, we introduce a refined type of taint analysis. During our data flow analysis, we collect all locations in persistent stores that are written to and can be controlled (tainted) by an adversary. If data is read from a persistent data store, the decision if the data is tainted or not is delayed to the end of the analysis. Eventually, when all taintable writings to persistent stores are known, the delayed decisions are made to detect second-order vulnerabilities. The intricacies of identifying the exact location within the persistent store the data is written to is approached with string analysis. Furthermore, sanitization through database lookups or checks for existing file names are recognized.

We implemented our approach in a prototype for static PHP code analysis since PHP is the most popular server-side scripting language on the Web with an increasing market share of 81.8% [28]. Note that our approach can be generalized to static code analysis of other languages by applying the techniques introduced in this paper to the data flow analysis of another language. We evaluated our approach by analyzing six popular real-world applications, including *OpenConf*, *HotCRP*, and *osCommerce*. Overall, we detected and reported 159 previously unknown second-order vulnerabilities such as *remote command execution* vulnerabilities in *osCommerce* and *OpenConf*. We also analyzed three web applications that were used during the evaluation of prior work in this area and found that previous work missed several second-order vulnerabilities, indicating that existing approaches do not handle such vulnerabilities correctly.

In summary, we make the following three contributions:

- We are the first to propose an automated approach to statically analyze second-order data flows through databases, file names, and session variables using string analysis. This enables us to detect second-order and multi-step exploitation vulnerabilities in web applications.
- We study the problem of second-order sanitization, a crucial step to lower the number of potential false positives and negatives.
- We built a prototype of the proposed approach and evaluate second-order data flows of six real-world web applications. As a result, we detect 159 previously unknown vulnerabilities ranging from XSS to *remote code execution* attacks.

2 Technical Background

In this section, we introduce the nature of second-order vulnerabilities and multi-step exploits. First, we examine data flow through persistent data stores and the difficulties of analyzing such flows statically. We then present two second-order vulnerabilities as motivating examples.

2.1 Persistent Data Stores

We define *persistent data stores* (PDS) as memory locations that are used by an application to store data. This data is available after the incoming request was parsed and can be accessed later on by the same application to reuse the data. The term *persistent* refers to the fact that data is stored on the web server's hard drive, although it can be frequently deleted or updated. Note that this definition also includes session data since information about a user's session is stored on the server and can be reused by an adversary. We now introduce three commonly used PDS by web applications.

2.1.1 Databases

Databases are the most popular form of PDS found in today's web applications. A database server typically maintains several databases that consist of multiple tables. A table is structured in columns that have a specific data type and length associated with them. Stored data is accessed via SQL queries that allow to filter, sort, or intersect data on retrieval. In PHP, an API for database interaction is bundled as a PHP extension that provides several built-in functions for the database connection, and the query and access of data.

In contrast to other PDS, *writing* and *reading* to a memory location is performed via the same built-in query function. SQL has different syntactical forms of writing data to a table. Listing 1 shows three different ways to perform the same query.

```
1 // specified write
2 INSERT INTO users (id,name,pass) VALUES (1,'admin','foo')
3 INSERT INTO users SET id = 1, name = 'admin', pass = 'foo'
4 // unspecified write
5 INSERT INTO users VALUES (1,'admin','foo')
```

Listing 1: Writing to the database table *users* in SQL.

While the first two queries explicitly define the column names, the third query does not. We refer to the first type as *specified write* and to the second type as *unspecified write*. Both types convey a difficulty for static

analysis of the query: a *specified write* reveals the column names where data is written to, but does not reveal if there are any other columns in the table that are filled with default values. This hinders the reconstruction of table structures when analyzing SQL queries of an application statically. An *unspecified write* tells us exactly how many columns exist, but does not reveal its names. When the columns are accessed later on by name, it is unclear which column was filled with which value. The same applies for read operations. A *specified read* reveals the accessed column names in a field list, whereas an *unspecified read*, indicated by an *asterisk* character, selects all available columns without naming them.

In PHP, the queried data is stored as a result resource. There are different ways to fetch the data from the result resource with built-in functions, as shown in Listing 2.

```
1 // numeric fetch
2 $row = mysql_fetch_row($res);      echo $row[1];
3 // associative fetch
4 $row = mysql_fetch_assoc($res);    echo $row["name"];
5 $row = mysql_fetch_object($res);   echo $row->name;
```

Listing 2: Fetching data from a database result resource.

Basically, *numeric* and *associative fetch* operations exist. The first method stores the data in a numerically indexed array where the index refers to the order of the selected columns. The *associative fetch* stores the data in an array indexed by column name. It is also possible to store the data in an object where the property names equals the column names. The key difference is that the *associative fetch* reveals the accessed column names while the *numeric fetch* does not.

All different combinations of writing, reading, and accessing data can occur within a web application. In certain combinations, it is not clear which columns are accessed without knowledge about the database schema. For example, when data is written *unspecified* and fetched associatively. In practice, however, we are often able to reconstruct the database schema from the source code (see Section 3.4.1 for details).

2.1.2 Session Data

A common way of dealing with the state-less HTTP protocol are *sessions*. In PHP, the `$_SESSION` array provides an abstract way of handling session data that is stored within files (default) or databases. A session value is associated with an alphanumerical key that represents the memory location. Note that the `$_SESSION` array needs to be treated like any other superglobal array in PHP and it can be accessed in any context of the application. As any other array, it can be accessed and modified dynamically, inter-procedurally, and it can have multiple key names. Besides the `$_SESSION` array and the deprecated `$HTTP_SESSION_VARS` array, the built-in functions `session_register()` and `session_decode()` can be used to set session data.

2.1.3 File Names

A common source for vulnerabilities is an unsanitized file name. Developers often overlook that the file name of an uploaded file can contain malicious characters and thus can be used as a PDS for an attack payload. For example, Unix file systems allow any special characters in file names, except for the slash and the null byte [12]. NTFS allows characters such as the single quote that can be used for exploitation [20]. For detecting second-order vulnerabilities, we need to determine paths where files with arbitrary names are located. The analysis of a file upload reveals to which path a file is written to and if the file is named as specified by the user. In PHP, a file that is submitted via a multi-part POST request is stored in a temporary directory with a temporary file name. The temporary and original file name is accessible in the superglobal `$_FILES` array. Furthermore, built-in functions such as `rename()` and `copy()` can be used by an application to rename a file on the server. Note that also directory names can be used as PDS, for example when created with the built-in function `mkdir()`.

2.1.4 Excluded PDS

There are less popular PDS that we do not include in our analysis. For example, data can be retrieved from a CGI environment variable, a configuration file, or from an external resource such as an FTP or SMTP server [5]. However, these PDS are used rarely in practice and decisions can only be made with preconfigured whitelists. We only consider PDS that are tainted by the application itself and not through a different channel. Analyzing the data flow through file content will be an interesting addition in the future. Here, the challenge is to determine to what part of a given file data is written to and from what part of the file data is read from because the structure of the data within the file is unknown.

Note that data stored via PHP's built-in functions `ini_set()` or `putenv()` only exists for the duration of the current request. At the end of the request, the environment is restored to its original state. Thus, they do not hold to our definition of a PDS.

2.2 Second-Order Vulnerabilities

A taint-style vulnerability occurs if data controlled by an attacker is used in a security-critical operation. In the data flow model, this corresponds to tainted data literally flowing into a sensitive sink within one possible data flow of the application. We classify a second-order vulnerability as a taint-style vulnerability where the data flows through one or more PDS. Here, the attack payload is first stored in a PDS and later retrieved and used in a sensitive sink. Thus, *two* distinct data flows require analysis: (i) source to PDS and (ii) PDS to sink.

In the following, we introduce two motivating examples with a payload stored in a PDS. In general, every combination of a source, sensitive sink, and a PDS is possible. Depending on the application's design, the flow of malicious data occurs within a single or multiple attack requests (e.g., when different requests for writing and reading are necessary). Finally, we introduce multi-step exploits as a subclass of second-order vulnerabilities.

2.2.1 Persistent Cross-Site Scripting

Cross-Site Scripting (XSS) [16] is the most common security vulnerability in web applications [22]. It occurs when user input is reflected to the HTML result of the application in an unsanitized way. It is then possible to inject arbitrary HTML markup into the response page that is rendered by the client's browser. An attacker can abuse this behavior by embedding malicious code into the response that for example locally defaces the web site or steals cookie information.

We speak of *Persistent Cross-Site Scripting* if the attacker's payload is stored in a PDS first, read by the application again, and printed to the response page. In contrast to *non-persistent* (*reflected*) XSS, the attacker does not have to craft a suspicious link and send it to a victim. Instead, all users of the application that visit the affected page are attacked automatically, making the vulnerability more severe. Furthermore, a *persistent XSS* vulnerability can be abused to spread an XSS worm [18, 26].

Listing 3 depicts an example of a *persistent XSS* vulnerability. The simplified code allows to submit a new comment which is stored in the table `comments` together with the name of the author. If no new comment is submitted, it lists all previously submitted comments that are fetched from the database. While the comment itself is sanitized in line 7 by the built-in function `htmlentities()` that encodes HTML control characters, the author's name is not sanitized in line 6 and thus affected by XSS. Note that if the source code is analyzed top-down, it is unknown at the point of the `SELECT` query if malicious data can be inserted into the table `comments` by an adversary.

```
1  if(empty($_POST['submit'])) {
2     // list comments
3     $res = mysql_query("SELECT author,text FROM comments");
4     foreach(mysql_fetch_row($res) as $row) {
5        $comment = mysql_fetch_array($row);
6        echo $comment['author'] . ': ' .
7        htmlentities($comment['text']) . "<br />";
8     }
9  }
10 else {
11    // add comment
12    $author = addslashes($_POST['name']);
13    $text = addslashes($_POST['comment']);
14    mysql_query("INSERT INTO comments (author, text)
15                 VALUES ('$author', '$text')");
16 }
```

Listing 3: Example for *second-order XSS* vulnerability.

2.2.2 Second-Order SQL Injection

A *SQL injection* (SQLi) [9] vulnerability occurs when a web application dynamically generates a SQL query with unsanitized user input. Here, an attacker can potentially inject her own SQL syntax to arbitrarily modify the query. Depending on the environment, the attacker can potentially extract sensitive data from the database, modify data, or compromise the web server.

In Listing 4, user supplied credentials are checked in line 6. If the credentials are valid, the session key `loggedin` is set to *true* and the user-supplied user name is saved into the session key `user`. In case the user-supplied data is invalid, the failed login attempt is logged to the database with the help of the user-defined `log()` function. Here, a *second-order SQLi* occurs: if an attacker registers with a malicious user name, this name is written to the session key `user` and on a second failed login attempt used in the logging SQL query.

```
1  function log($error) {
2     $user = $_SESSION['user'];
3     mysql_query("INSERT INTO logs (error, user)
4                  VALUES ('$error', '$user')");
5  }
6  if(validAuth($_POST['user'], $_POST['pass']) {
7     $_SESSION['loggedIn'] = true;
8     $_SESSION['user']     = $_POST['user'];
9  }
10 else {
11    log('Failed login attempt');
12 }
```

Listing 4: Example for *second-order SQLi* vulnerability.

2.2.3 Multi-Step Exploitation

Within a second-order vulnerability, the first order (e. g., safe writing of user input into the database or a file path) is not a vulnerability by itself. However, *unsafe* writing can lead to other vulnerabilities. We define a multi-step exploit as the exploitation of a vulnerability in the second order that requires the exploitation of an unsafe writing in the first order. Thus, a multi-step exploit is a subclass of a second-order vulnerability and it can drastically raise the severity of the first vulnerability.

Since we only consider databases, sessions, and file names as PDS in our analysis, the following vulnerabilities are relevant:

- *SQLi*: A SQLi in an `INSERT` or `UPDATE` statement leads to a full compromise of all columns in the specified table. Furthermore, a SQLi in a `SELECT` query allows arbitrary data to be returned.
- *Path traversal*: A *path traversal* vulnerability allows to change the current directory of a file operation to another location. Arbitrary file names can be created in arbitrary locations if a *path traversal* vulnerability affects the renaming or creation of files.
- *Arbitrary file write*: An *arbitrary file write* vulnerability can modify or create a new session file, leading to the compromise of all session values.

3 Detecting Second-Order Vulnerabilities

In the following, we describe our approach to automatically detect second-order vulnerabilities via static code analysis. For this purpose, we extended our prototype *RIPS* [6] that uses *block summaries* [30]. In this section, we first briefly review the used data flow and taint analysis approach of RIPS (Sections 3.1 and 3.2). Afterwards, we explain our novel additions for detecting second-order vulnerabilities and multi-step exploits (Sections 3.3–3.5).

3.1 Data Flow Analysis

RIPS leverages a context-sensitive, intra- and interprocedural data flow analysis. We use basic block, function, and file *summaries* [30] for efficient, backwards-directed data flow analysis [6]. First, for each PHP file's code, a *control flow graph* (CFG) consisting of connected basic blocks is generated. Definitions of functions, classes, and methods within the code are extracted. Then, every CFG is analyzed top-down by simulating the connected basic blocks one by one. A block edge that links two connected basic blocks is simulated as well to identify data sanitization.

During the simulation of one basic block, all assigned data is transformed into *data symbols* that we will introduce later. The flow of the data is inferred from these symbols and summarized in a *block summary* [30] that maps data locations to assigned data. The return results and side-effects (e.g., data assignment or sanitization) of called built-in functions are determined by a precise simulation of over 900 unique functions.

If a user-defined function is called within a basic block, its CFG is generated and all basic blocks are simulated. Based on these block's summaries, the data flow within the function is determined by analyzing `return` statements in a similar way to taint analysis (see Section 3.2). The results are stored in a *function summary*. This summary is used for each call of the user-defined function, while return values, global variables, and parameters are adjusted to the callee's arguments and environment context-sensitively. When all basic blocks of a file's CFG are simulated, a *file summary* is generated in a similar way to functions that is used during file inclusion.

Data and its access within the application's code is modeled by so called *data symbols* [6]:

- `Value` represents a static `"string"`, integer, float, or a resolved `CONSTANT`'s value. Defined constant values are stored in the environment.
- `Variable` represents a `$variable` by its name.
- `ArrayDimFetch` represents the access of an array (`$array[k]`) and extends the `Variable` symbol with a *dimension* (`k`). The dimension lists the fetched array keys in form of data symbols.
- `ArrayDimTree` represents a newly declared array or the assignment of data to one array key (`$array[k] = $data`). It is organized in a tree structure. The array keys are represented by array edges that point to the assigned data symbol. The `ArrayDimTree` symbol provides methods to add or fetch symbols by a *dimension* that is compared to the tree's edges.
- `ValueConcat` represents the concatenation of two or more data symbols (`$a.$b`). Two consecutive `Value` symbols are merged to one `Value` symbol.
- `Multiple` is a container for several data symbols. It is used, for example, when a function returns different values depending on the control flow or PHP's *ternary* operator is used (`$c ? $a : $b`).

During data flow analysis, one or more *sanitization tags* can be added to a data symbol, for example if sanitization is applied by built-in functions such as `addslashes()` or `htmlentities()`. Each sanitization tag represents one context, for example, a *single-quoted SQL value* or a *double-quoted HTML attribute*. A symbol can be sanitized against one context, but be vulnerable to another. The tags are removed again when built-in functions such as `stripslashes()` or `html_entity_decode()` are called. Furthermore, information about encoding is added to every data symbol.

3.2 Context-Sensitive Taint Analysis

The goal is to create a vulnerability report, whenever a tainted data symbol δ flows into a sensitive sink. Our implementation is performed with 355 sensitive built-in functions of PHP. If a call to a sink is encountered during block simulation, its relevant arguments are analyzed. First, the argument is transformed into a data symbol. If the symbol was defined within the same basic block, it is inferred from the block summary. Then, the symbol is looked up in the block summary of every previous basic block that is linked with a block edge to the current basic block. If the lookup in the block summary succeeds, the inferred symbol is fetched. The dimension of an `ArrayDimFetch` symbol is carried until a mapping `ArrayDimTree` symbol is found. The backwards-directed symbol lookup continues for each linked basic block and stops if a symbol of type `Value` is inferred or the beginning of the CFG is reached. At this point, all resolved symbols are converted to strings in order to perform context-sensitive string analysis [6]. The symbols `Value` and `Boolean` are converted to their representative string values. Data symbols of sources are mapped to a *Taint ID* (TID) that is used as string representation.

Next, each string is analyzed. The location of the TIDs within the markup is determined to precisely detect the context. For complex markup languages such as HTML

or SQL, a markup parser is used. With the help of the *sanitization tags* and encoding information of the linked data symbol, we check if the symbol is sanitized correctly according to its context. If a TID is found that belongs to an unsanitized source regarding the current context, a vulnerability report is generated. Unsanitized parameters or global variables are added to the function's summary as *sensitive parameter* or *global*. These are analyzed in the context of each function call.

3.3 Array Handling

By manually analyzing the code of the most popular PHP applications [28], we empirically found that a common way to write data into a database is by using arrays. An example is shown in Listing 5. In line 9 and 10, the array's *key* defines the table's column and the array's *value* stores the data to write. The separated array values are joined to a string again by using the built-in function `implode()` (lines 2/3). Based on this observation, we redesigned the handling of arrays by adding new data symbols. As a side effect, the handling of fetched database results in form of an array and the handling of the superglobal `$_SESSION` array is significantly improved.

```
1  function insert($table, $array) {
2     $fields = implode(",", array_keys($array));
3     $values = implode("','", $array);
4     mysql_query("INSERT INTO {$table}
5                  (".$fields.") VALUES ('".$values."')");
6  }
7
8  $new_user = array(
9     "name" => addslashes($_POST['name']),
10    "pass" => md5($_POST['pass']),
11 );
12 insert("users", $new_user);
13 // INSERT INTO users (name,pass) VALUES ('X', '123abc...')
```

Listing 5: Using arrays to write data to a database.

We model the popular built-in function `implode()` by adding the data symbol `ArrayJoin`. With the help of this symbol, it is possible to keep track of the *delimiter* that is used to join strings. If the symbol is inferred to an `ArrayDimTree` symbol, a `ValueConcat` symbol is created that joins all symbols of the `ArrayDimTree` symbol with the stored delimiter symbol.

Furthermore, we introduce the new symbol `ArrayKey`. It is used when the *key* of an array is explicitly accessed, such as in the loop `foreach($array as $key => $value)`. It is handled similar to the `Variable` symbol and is associated with the array's name. If the `ArrayKey` symbol is inferred into an `ArrayDimTree` symbol during data flow or taint analysis, a `Multiple` symbol containing all edges' symbols is returned. Built-in functions, such as `array_keys()` and `array_search()`, return all or parts of the available keys in an array and can be modeled more precisely with the `ArrayKey` symbol.

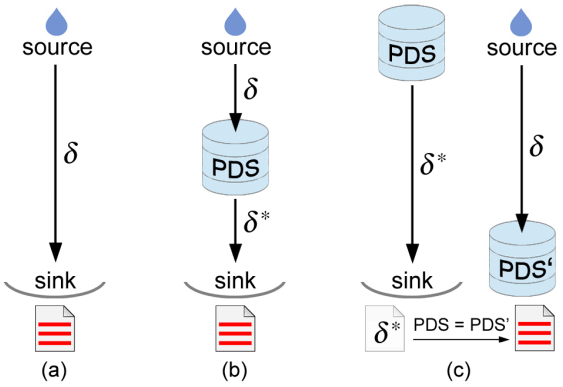


Figure 1: Data flow model of a conventional (a) and a second-order (b, c) vulnerability.

3.4 PDS-centric Taint Analysis

We now introduce our novel approach to detect second-order vulnerabilities. The data flow is illustrated in Figure 1 (b). Contrarily to a conventional taint-style vulnerability as shown in Figure 1 (a), a source flows into a PDS before it flows from the PDS into a sensitive sink. We model the data that is read from a PDS by new data symbols δ^* that hold information about their origin.

During code analysis, *taintable* PDS are identified. They are stored together with the minimum set of applied sanitization and encoding tags of the tainting data symbol δ. If one of the new data symbols δ^* is encountered unsanitized during the taint analysis of a sensitive sink, a vulnerability report is created if its originating PDS was identified as taintable.

If the PDS is not known as taintable, a *temporary* vulnerability report is created, as shown in Figure 1 (c). The report is connected to the data symbol δ^*. At the end of the code analysis, we decide if the data symbol originates from a taintable PDS by comparing its origin to all collected taintable PDS.

In the following, we introduce the analysis of writings to different PDS. Furthermore, our new data symbols δ^* are introduced that model the reading and access of data that is stored in PDS.

3.4.1 Databases

Modeling the data flow through databases is a complex task, mainly due to the large API that is available for databases and the usage of a query language. First, our prototype tries to obtain as much knowledge of the SQL schema as possible. Then we try to reconstruct all SQL queries during SQL injection analysis of 110 built-in query functions. Finally, the type of operation is determined, as well as the targeted table and column names. The access of data is modeled by new data symbols.

Preparation During the initialization of our tool, we collect all files with a *.sql* extension. All available `CREATE TABLE` instructions within these files are parsed so that we can reconstruct the database schema, including all table and column names as well as column types and length. If no schema file is found, each PHP file in the project is searched via regular expression. The knowledge of the database schema improves precision when data is read in an *unspecified* way, or when data is sanitized by the column type or length.

Writing A write operation to a database is detected if the SQL parser identifies an `INSERT`, `UPDATE`, or `REPLACE` statement. By tokenizing the SQL query, we determine the targeted table's name, all specified column names, and their corresponding input values. In case of an *unspecified write*, the parser makes use of the database schema. If an input value of a column contains a TID (see Section 3.2), the affected column and table name is marked as *taintable* together with the linked source symbol and its sanitization tags.

Reading If the SQL parser encounters a `SELECT` statement, we try to determine all selected column and table names. Multiple table names can occur if tables are *joined* or *unioned*. Alias names within the query are mapped and resolved. In case of uncertainty, the parser makes use of the database schema. Finally, a new `ResourceDB` symbol is mapped to the analyzed query function as return value. This symbol holds information about all selected column names in a numerical hash map and its corresponding table names.

Access In PHP, database result resources are transformed into arrays by built-in fetch functions (refer to Listing 2). We ignore the mode of access and let 89 configured fetch functions return a `Variable` symbol with the name of the resource. When an `ArrayDimFetch` symbol accesses the result of these fetch functions, it is inferred to the corresponding `ResourceDB` symbol. In this case, the carried dimension of the `ArrayDimFetch` symbol is evaluated against the available column names in the `ResourceDB` symbol. If the *asterisk* character is contained in the column list and the dimension is numerical, the database schema is used to find the correct column name. Otherwise, if the dimension equals a column name in the field list, a new `DataDB` symbol is returned that states which column of which table is accessed.

Sanitization Certain implicit sanitization is considered when dealing with SQL. If a column is compared to a static value within a `WHERE` clause in a `SELECT` statement, the return value for this column is sanitized. In this case, the static value is saved within the `ResourceDB` symbol and mapped to the column as return value. Furthermore, a sanitization tag for the used quote type is removed when data is updated or inserted to the database because one level of escaping is lost during writing.

3.4.2 Session Keys

The analysis of session variables does not require a complex markup parser or new data symbol. Instead, session data is handled similar to other global arrays. Taintable session keys are stored during the analysis phase.

Writing If data is assigned to a `Variable` or `ArrayDimFetch` symbol during block simulation and the symbol's name is `$_SESSION`, the assigned data is analyzed via taint analysis. If the assigned data is tainted, its resolved source symbol is stored into an `ArrayDimTree` symbol in the environment, together with the dimension of the `$_SESSION` symbol. This way, an `ArrayDimTree` is built with all taintable dimensions of the session array that link to the tainted source symbols and their corresponding sanitization tags.

Reading The access to session data is modeled by `ArrayDimFetch` symbols with the name `$_SESSION` and requires no modification. During taint analysis inside a user-defined function, session variables are handled as global variables. They are added to the function summary and they are inspected for each function call in a context-sensitive way. This avoids premature decisions about the taint status inside a function if the session key is overwritten before the function is called. Just as for a `DataDB` symbol, a *temporary* vulnerability report is created if a `$_SESSION` variable taints a sensitive sink.

3.4.3 File Names

To detect taintable file names, we collect file paths a user can write to. For this purpose, new data symbols model directory resources and their accesses. Whenever a path is reconstructed only partially, we use the same approach as in file inclusion analysis. Here, a regular expression is created and mapped to all available paths that were detected when loading the application files.

Writing To detect a file name manipulation with user input, we analyze 27 built-in functions such as `copy()`, `rename()`, and `file_put_contents()`. Additionally, file uploads with `move_uploaded_file()` are analyzed. Note that at the same time these built-in functions are sensitive sinks and generate vulnerability reports such as an *arbitrary file upload* vulnerability. The path argument is analyzed by conventional context-sensitive string analysis. If the path is tainted, we store it with its prefix as taintable. When no prefix is present, the file path of the currently analyzed file is taken. Additionally, if the source is not sanitized against *path traversal* attacks, all paths are assumed as taintable and a flag is set during analysis accordingly.

Reading We handle three different ways of opening a directory with PHP's built-in functions. First, we model the built-in function `scandir()` that returns an array, listing all files and directories within a specified path.

Second, we model the built-in function `glob()` that also returns an array that lists all files and directories specified by a pattern. We transform the pattern into a regular expression by substituting the pattern characters `*` and `?` into regular expression equivalents. Third, we model the built-in function `opendir()` which returns a *directory handle*. For all mentioned built-in functions, we reconstruct the opened path by string analysis and return a `ResourceDir` symbol that stores the path's name.

Access The returned result of `scandir()` and `glob()` is accessed by an array key. Since we do not know neither the amount nor the order of files in a directory, we return a `DataPath` symbol whenever a `ResourceDir` symbol is inferred from an `ArrayDimFetch` symbol, regardless of its dimension. For this purpose, we let the built-in function `readdir()` that is supposed to read an entry of a *directory handle* return an `ArrayDimFetch` symbol with an arbitrary dimension and the name of the directory handle. It is inferred to a `DataPath` symbol when the trace of the `ArrayDimFetch` symbol results in a `ResourceDir` symbol.

Sanitization In order to model sanitization that checks if a given string is a valid file name, 11 built-in functions such as `file_exists` and `is_file()` are simulated. We modified the sanitization check in a way that these functions only sanitize if there is no taintable file path found. For this purpose, a flag is set during taint analysis if sanitization of a source by file name is detected. The flag issues only a temporary vulnerability report that is revised at the end of the analysis regarding the ability to taint a file path.

3.4.4 Multi-Step Exploits

In order to detect multi-step exploits, we store all table names of all writing SQL queries that are affected by SQLi. Furthermore, we set a flag during the analysis process if an *arbitrary file write* or *arbitrary file rename* vulnerability is detected. At the end of the analysis, when the taint decision is made for data that comes from a PDS, multi-step exploit reports are added to the initial vulnerability. This is done for all vulnerabilities that rely on a `DataDB` symbol that is not tainted through second-order but which table name is affected by SQLi. Also, a multi-step exploit is reported if a `DataDir` symbol occurs and the flag for a *file rename* vulnerability was set. All session data is treated as tainted if an *arbitrary file write* vulnerability was detected. Additionally, any *local file inclusion* vulnerability is extended to a *remote code execution* if a file write or upload feature is detected.

Moreover, a SQLi vulnerability within a `SELECT` query returns a `DataDB` symbol with a *taint* flag. This flag indicates that all accessed columns are taintable by modifying the `SELECT` query during an attack. Thus, all columns of the `DataDB` symbol are taintable.

3.5 Inter-procedural PDS Analysis

We optimized the inter-procedural analysis to refine our string analysis results. Function summaries offer a high performance but they are also inflexible for functions with dynamic behavior. Thus, they can weaken the static reconstruction of dynamically created strings.

3.5.1 Multiple Parameter Trace

As we illustrated in Listing 5, modern applications often define wrapper functions for PDS access where more than one parameter is used within one sensitive sink. In this case, the approach of storing each parameter together with its prefixed and postfixed markup, and the corresponding vulnerability type as *sensitive parameter* in the function summary, is error-prone. When a call to this function occurs, the approach swaps the parameter symbol with the argument of the function call and traces it for user input. While this approach works fine for vulnerability detection, it leads to imprecision when it comes to string reconstruction. Because each argument is traced separately but both are used in the same sink, the result of one trace is missing in the result of the other trace. In Listing 5, for example, the table name is missing in the reconstructed query while the data is reconstructed from the `$new_user` array.

To circumvent this problem, we refined this approach for sinks that execute SQL queries or open file paths within a user-defined function. If multiple parameters or global variables are involved, all symbols are combined to one `ValueConcat` symbol. Then this symbol is stored in the function summary and analyzed for each function call. This way, each parameter is traced within one analysis and all results are present at the same time.

3.5.2 Mapping Returned Resources

Working with function summaries is very efficient when it comes to performance because each function only requires a single analysis on the first call. For every other call, the function summary is reused. However, a user-defined function might return a resource that has different properties for each call. For example, a `SELECT` query that embeds the parameter of an user-defined function as the table name returns a different `ResourceDB` symbol for every call, depending on the function's argument. If the resource is returned by the user-defined function, its symbol's properties change for every different function call.

As a solution, we add empty `ResourceDB` symbols to the function summary's set of return values for user-defined functions with dynamic SQL queries. Once the sensitive parameters are analyzed and the queries are reconstructed, a copy of these symbols is updated with the table and column information and used as returned data.

4 Evaluation

For evaluating our approach, we selected six real-world web applications. We chose the conference management systems *OpenConf 5.30* and *HotCRP 2.61* for their popularity in the academic field and *osCommerce 2.3.3.4* for its large size. Furthermore, we evaluated the follow-up versions of the most prominent software used in related work [3, 11, 30, 31]: *NewsPro 1.1.5*, *MyBloggie 2.1.4*, and *Scarf 2007-02-27*.

A second-order vulnerability consists of two data flows: tainting the PDS and tainting the sensitive sink. We evaluated our prototype for both steps and present the *true positives* (TP) and *false positives* (FP) in this section. In addition, we discuss the root cause for *false negatives* (FN) and outline the limitations of our approach.

4.1 PDS Usage and Coverage

To obtain an overview of the usage of PDS in web applications, we manually evaluated the total amount of different memory locations. Note that these numbers do not reflect how often one memory location is used at runtime. Then, we evaluated the ability to taint these memory locations by an application's user and compared it to the detection rate of our prototype. A PDS is defined as *taintable* if it can contain at least one of the following characters submitted by an application user: `\<>'"`. In total, we manually identified 841 PDS of which 23% are taintable. Our prototype successfully detected 71% of the taintable PDS with a false discovery rate of 6%.

4.1.1 Databases

Our implementation successfully recovered the database schema for all tested applications during the initialization phase. For evaluation, we categorized all available columns in the application's database schema by declared data type. Only columns with a *string* type, such as `VARCHAR` or `TEXT`, are of interest because they can store tainted data. As shown in Table 1, we found that on average about half of the columns are not taintable due to numeric data types such as `INT` and `DATE`.

Table 1: Column types in selected applications.

Software	Tables	Columns	Num	String
osCommerce	50	331	193	138
HotCRP	29	217	142	75
OpenConf	18	129	48	81
NewsPro	8	43	18	25
Scarf	7	37	22	15
MyBloggie	4	24	10	14
Total	116	781	55%	45%

We then carefully fuzzed a local instance of each application manually with common attack payloads in order to determine which columns of type *string* are taintable. Furthermore, we observed which columns were reported by our prototype implementation as taintable when the schema is available and when not. The results are compared in Table 2. Among the columns with a string type, 53% are taintable. As a result, only 24% of all available columns are not sanitized by the application or the columns' data type.

Table 2: Taintable columns in selected applications.

		Schema		No schema	
Software	Taintable	TP	FP	TP	FP
osCommerce	63	55	4	55	37
HotCRP	43	27	1	27	3
OpenConf	47	16	1	16	4
NewsPro	12	12	0	12	0
Scarf	10	10	1	10	3
MyBloggie	9	9	0	9	0
Total	184	70%	5%	70%	27%

For the rather old and simple applications, all taintable columns were detected by our prototype. The modern and large applications often use loops to construct dynamic SQL queries where reconstruction is error-prone. Overall, we detected 70% of all taintable columns. When the database schema is known, 5% of our reports are FP. The root cause is path-sensitive sanitization of data that is written to the database—a sanitization that our current prototype is not able to detect yet. The false discovery rate is higher if the database schema of an application is not found. In this case, a static analysis tool cannot reason about data types within the database and may flag columns of numeric data type as taintable.

4.1.2 Sessions

To obtain a ground truth for our evaluation, we again manually assessed the applications' code for all accessed keys of the superglobal `$_SESSION` array. Dynamic keys were reconstructed and keys in multi-dimensional arrays were counted multiple times. Then, we manually examined which session keys are taintable by the application's user and compared this to the analysis result generated by our prototype implementation. As shown in Table 3, we found that only 12% of the 52 identified session keys are taintable within our selected applications.

Our prototype correctly detected all taintable session keys. One FP occurred because the sanitized email address of a user is written to the session after it is fetched from the database. This FP is based on the previously introduced FP in identifying taintable columns. A custom session management in *osCommerce* led to exclusion from our evaluation.

Table 3: Taintable session keys in selected applications.

Software	Keys	Taintable	TP	FP
HotCRP	29	2	2	0
OpenConf	14	2	1	0
NewsPro	2	1	1	0
Scarf	4	0	0	1
MyBloggie	3	1	1	0
Total	52	12%	83%	16%

4.1.3 File Names

To evaluate the features that allow an application's user to alter a file name, we again manually assessed each application for file upload, file creation, and file rename features and counted the different target paths to obtain a ground truth. Next, we counted the collected taintable path names reported by our prototype. The results are shown in Table 4.

Table 4: Taintable paths in selected applications.

Software	Paths	Taintable	TP	FP
osCommerce	2	2	2	0
HotCRP	1	0	0	0
OpenConf	1	0	0	1
NewsPro	1	0	0	0
Scarf	1	1	1	0
MyBloggie	2	2	2	0
Total	8	63%	100%	16%

We found at least one feature in each of the application's source code to create a new file. However, half of the applications sanitize the name of the file before creating it. Our prototype detected all taintable path names. One FP occurred for *OpenConf*, where uploaded files are sanitized in a path-sensitive way.

Interestingly, a file upload in *Scarf* is based on a second-order data flow. The name of the uploaded file is specified separately and stored as a configuration value in the database before it is read from the database again and the file is copied. Because no sanitization is applied, an administrator is able to copy any file to any location of the server's file system which leads to *remote code execution*. This critical vulnerability was missed in previous work that also used this application for evaluating their approach [3, 31].

4.2 Second-Order Vulnerabilities

We evaluated the ability of our prototype to detect second-order vulnerabilities. Reports of first-order vulnerabilities are ignored for now. Our prototype reported a total of 159 valid second-order vulnerabilities with a false discovery rate of 21% (see Table 5 for details). In summary, 97% of the valid reports are *persistent XSS* vulnerabilities where the payload is stored in the database. Five *persistent XSS* vulnerabilities are caused by session data or file names. This is closely related to the fact that 94% of all taintable PDS we identified are columns in database tables (see Section 4.1) and sensitive sinks such as `echo` are one of PHP's most prominent built-in features [10].

Table 5: Evaluation results for selected applications.

Software	Files	LOC	TP	FP	FN
osCommerce	570	66 381	97	29	6
HotCRP	74	40 339	1	1	0
OpenConf	121	20 404	16	4	0
NewsPro	23	5 077	7	1	0
Scarf	19	1 686	37	8	3
MyBloggie	58	9 485	1	0	0
Total	865	143 372	159	43	9
Average	144	23 895	79%	21%	

Our evaluation revealed that second-order vulnerabilities are highly critical. Next to *persistent XSS* and file vulnerabilities, we detected various *remote code execution* vulnerabilities in *osCommerce*, *OpenConf*, and *NewsPro*. In the following, we introduce two selected vulnerabilities to illustrate the complexity and severity of real-world second-order vulnerabilities. It is evident that these vulnerabilities could only be detected with our novel approach of analyzing second-order data flows.

4.2.1 Second-Order LFI to RCE in OpenConf

OpenConf is a well-known conference management software used by many (academic) conferences. Our prototype found a *second-order local file inclusion* vulnerability in the user-defined `printHeader` function that leads to *remote command execution*. The relevant parts of the affected file *include.php* is shown in Listing 6.

```
1  function printHeader($what, $function="0") {
2     require_once $GLOBALS['pfx'] .
3        $GLOBALS['OC_configAR']['OC_headerFile'];
4  }
5
6  $r = mysql_query("SELECT `setting`, `value`, `parse`
7                    FROM `" . OCC_TABLE_CONFIG . "`");
8  while ($l = mysql_fetch_assoc($r)) {
9     $OC_configAR[$l['setting']] = $l['value'];
10 }
11 printHeader();
```

Listing 6: Simplified *include.php* of *OpenConf*.

When looking at the code, it does not reveal any vulnerability. Whenever the code is included, settings are loaded from the database and the user-defined function `printHeader()` is called. This function includes a configured header file and prints some HTML.

```
1  function updateConfigSetting($setting, $value) {
2     $q = "UPDATE `" . OCC_TABLE_CONFIG . "`
3           SET `value`='" . safeSQLstr(trim($value)) . "'
4           WHERE `setting`='" . safeSQLstr($setting) . "'";
5     return(ocsql_query($q));
6  }
7
8  foreach (array_keys($_POST) as $p) {
9     if (preg_match("/^OC_[\w-]+$/", $p)) {
10       updateConfigSetting($p, $_POST[$p]);
11    }
12 }
```

Listing 7: Simplified code to change settings in *OpenConf*.

However, as shown in Listing 7, it is possible for a privileged chair user to change any configuration setting. The configuration page does not specify an input field to change the *headerFile* setting. Nonetheless, by adding the key *OC_headerFile* to a manipulated HTTP POST request, the setting is changed. The loop over the submitted keys of the `$_POST` array in Listing 7, line 8, as well as the loop over the `$OC_configAR` in Listing 6, line 9, shows once again how important it is to track the taint status of PHP's array keys precisely.

A chair member can now include any local file of the system to the output. Additionally, because the software allows to upload PDF files to the server, our prototype added a multi-step exploit report. Indeed, if a PDF file containing PHP code is uploaded to the server and the *headerFile* setting is pointed to that PDF, arbitrary PHP code is executed. Moreover, our tool reported a SQL injection vulnerability that is accessible to unprivileged users. This allows any visitor to extract the chair's password hash (salted SHA1) from the database.

4.2.2 Second-Order RCE in NewsPro

Utopia NewsPro is a blogging software and was used in previous work for evaluation [29–31]. Our prototype reported a *second-order code execution* vulnerability in the administrator interface. Here, a user is able to alter the template files of the blog. The simplified code is shown in Listing 8.

```
1 $tempid = (int)$_POST['tempid'];
2 $template = mysql_real_escape_string($_POST['template']);
3 $updateTemplate = mysql_query("UPDATE `unp_template`
4         SET template='$template' WHERE id='$tempid'");
```

Listing 8: Simplified code to change the template in *NewsPro*.

The template code is read from the database in various places of the source code with help of the user-defined function `unp_printTemplate()` (see Listing 9). First, this function writes the template's code to a cache array (line 6) and then returns it from this array again. The example demonstrates the importance of inter-procedural analysis and array handling.

```
1  function unp_printTemplate($template) {
2     global $templatecache, $DB;
3     $getTemplate = mysql_query("SELECT name,template
4       FROM `unp_template` WHERE name='$template' LIMIT 1");
5     while ($temp = mysql_fetch_array($getTemplate)) {
6        $templatecache[$template] = $temp['template'];
7     }
8     return addslashes($templatecache[$template]);
9  }
10 eval('$headlines_displaybit = "' .
11        unp_printTemplate('headlines_displaybit').'";');
```

Listing 9: Simplified *Remote Code Execution* vulnerability in *NewsPro*.

At the call-site, the fetched template is evaluated with PHP's `eval` operator that executes PHP code (line 10). The template's code is escaped (line 8), however, the double-quoted value of the evaluated variable `$headlines_displaybit` allows to execute arbitrary PHP code using curly syntax. By adding the code {${system(id)}} to a template, the system command *id* is executed. Note that related work missed to detect this vulnerability, which is also present in prior versions.

4.3 Multi-Step Exploits

Our prototype reported two *arbitrary file upload* vulnerabilities and 14 SQL injection vulnerabilities. Because these vulnerabilities affect a storage operation, the stored data can be manipulated during multi-step exploitation. Our prototype found 14 valid multi-step exploits and a single FP as shown in Table 6.

Table 6: Reported multi-step exploits in selected applications.

Software	File TP	SQLi TP	SQLi FP	Multi-Step TP	Multi-Step FP
osCommerce	1	3	0	3	0
HotCRP	0	1	7	0	1
OpenConf	0	4	1	1	0
NewsPro	0	6	0	9	0
Scarf	1	1	0	1	0
MyBloggie	0	5	0	0	0
Total	2	20	8	14	1
Average	100%	71%	29%	93%	7%

All detected multi-step exploits consist of two steps and no *third-order* vulnerabilities were detected within our selected applications. In the following, we examine two multi-step exploits in *osCommerce* that lead to *remote command execution* to illustrate that these vulnerabilities can only be detected with our novel approach of analyzing multi-step exploits.

4.3.1 Multi-Step RCE in osCommerce

OsCommerce is a popular e-commerce software. For one of three reported SQLi vulnerabilities in *osCommerce*, our prototype additionally reported a *multi-step remote code execution* exploit. The SQLi is located in the backup tool of the administrator interface and shown in Listing 10. Here, a SQL file is uploaded to restore a database backup. Since the name of the uploaded file is later used unsanitized in a SQL query, an attacker is able to insert any data into the `configuration` table by uploading a SQL file with a crafted name. This enables another, more severe vulnerability: the table `configuration` stores a `configuration_value` and a `configuration_title` for each setting. Furthermore, a `use_function` can be specified optionally to deploy the configuration's value.

```
1 $sql_file = new upload('sql_file');
2 $read_from = $sql_file->filename;
3 tep_db_query("insert into " . TABLE_CONFIGURATION .
4    " values (null, 'Last Database Restore', 'DB_RESTORE',
5    '" . $read_from . "', 'Last database restore file',
6    '6', '0', null, now(), '', '')");
```

Listing 10: Simplified code of the *backup.php* file in *osCommerce* shows a SQLi through a file name.

When the list of configuration values is loaded from the database, the function name specified in the `use_function` column is called with the `configuration_value` as argument (see Listing 11, line 5). An attacker can abuse the SQLi to insert an arbitrary PHP function's name, such as `system`, to the column `use_function` and insert an arbitrary argument, such as *id*, to the column `configuration_value`. When loading the configuration list, the specified function is fetched and called with the specified argument that executes the system command *id*.

```
1 $conf_query = tep_db_query("select configuration_id,
      configuration_title, configuration_value,
      use_function from " . TABLE_CONFIGURATION . " where
      configuration_group_id = '" . (int)$gID . "'");
2 while ($configuration = tep_db_fetch_array($conf_query)) {
3    if (tep_not_null($configuration['use_function'])) {
4       $use_function = $configuration['use_function'];
5       $cfgValue = call_user_func($use_function,
6                   $configuration['configuration_value']);
```

Listing 11: Simplified code of the *configuration.php* file in *osCommerce* demonstrates a *multi-step RCE*.

4.3.2 Sanitization Bypass in osCommerce

Another *multi-step RCE* exploit was reported in *osCommerce* that involves a sanitization bypass. The previously mentioned backup tool of the administrator interface allows to specify a local ZIP file that is unpacked via the system command *unzip*. Here, the target file name is specified as an argument in the command line if the specified file name exists on the file system. The simplified code is shown in Listing 12.

```
1 if (file_exists(DIR_FS_BACKUP . $HTTP_GET_VARS['file'])) {
2    $restore_file = DIR_FS_BACKUP . $HTTP_GET_VARS['file'];
3    exec(LOCAL_EXE_UNZIP . ' ' . $restore_file . ' -d ' .
         DIR_FS_BACKUP);
4 }
```

Listing 12: A dynamically constructed system command in *osCommerce* includes the name of an existing file.

An attacker can bypass this check by abusing one of the file upload functionalities in *osCommerce*. By uploading a file with the name *;id;.zip* and afterwards specifying this file as backup file, the command *id* is executed. The semicolons within the file name terminate the previous *unzip* command and introduce a new command.

4.4 False Positives

Our prototype generated 43 false second-order vulnerability reports, leading to a false discovery rate of 21% for our selected applications. All false positives are based on the fact that our prototype is not able to detect path-sensitive sanitization. Thus, *persistent XSS* was reported in *Scarf* and *HotCRP* that are based on email addresses stored in the database. Our prototype erroneously identified these columns as taintable (see Section 4.1.1). The same error applies to a paper format in *OpenConf* which leads to four false positives. A user-defined sanitization function using path-sensitive sanitization based on its argument lead to 29 false *persistent XSS* reports in *osCommerce*. A false multi-step exploit was reported in *HotCRP* caused by a false SQLi report. By performing a path-sensitive sanitization analysis, these false positives can be addressed in the future.

4.5 False Negatives

Evaluating false negatives is an error-prone task because the actual number of vulnerabilities is unknown. Furthermore, no CVE entries are public regarding second-order vulnerabilities in our selected applications. However, it is possible to test for false negatives that stem from insufficient detection of taintable PDS. By pre-configuring our implementation with the taintable PDS we identified manually, we can compare the amount of detected second-order vulnerabilities with the number of reports when PDS are analyzed automatically.

As a result, only six previously missed *persistent XSS* in *osCommerce* were reported. Additionally, another taintable session key in *OpenConf* was reported, although the key does not lead to a vulnerability. Furthermore, we manually inspected the source code of the applications and observed that our SQL parser needs improvement. Three false negatives occurred in *Scarf* because our parser does not handle SQL string functions such as `concat()`. More complex SQL instructions might lead to further false negatives but are used rarely.

4.6 Performance

We evaluated our prototype with the implementation of our approach to detect second-order vulnerabilities (+SO) and without it (-SO). Our testing environment was equipped with an Intel i7-2600 CPU with 3.4 GHz and 16 GB of memory. The amount of memory consumption (M, in megabytes), scan time (T, in seconds), and second-order vulnerability reports (R) for our selected applications are given in Table 7.

Table 7: Performance results for selected applications.

	-SO Analysis		+SO Analysis		
Software	M[mb]	T[s]	M[mb]	T[s]	R
osCommerce	834	134	846	213	129
HotCRP	752	186	775	345	3
OpenConf	528	33	523	47	21
NewsPro	50	1	50	3	17
Scarf	39	1	40	14	46
MyBloggie	87	7	87	11	1
Total	2290	362	2321	633	217
Average	382	60	387	106	36

While the memory consumption does not increase significantly by adding second-order analysis, the average scan time increases by 40%. Note, however, that this includes 217 processed vulnerability reports the prototype would have missed without the additional second-order analysis. Furthermore, we believe that a total scan time of less than 11 minutes for our selected applications is still reasonable.

5 Related Work

Web applications are widely used in the modern Web and as a result, security analysis of such applications has attracted a considerable amount of research. We now review related work in this area and discuss how our approach differs from previous approaches.

Dynamic Analysis There are many different dynamic approaches to perform a security analysis of a given web application. For example, Apollo [1] leverages symbolic and concrete execution techniques in combination with explicit-state model checking to perform persistent state analysis for session variables in PHP. Sekar proposes syntax- and taint-aware policies that can accurately detect and/or block most injection attacks [23]. However, such approaches are typically limited to simple types of taint-style vulnerabilities.

There are also dynamic approaches to detect second-order vulnerabilities. For example, McAllister et al. present a blackbox scanner capable of detecting *persistent XSS* [19]. Ardilla [14] aims at detecting both SQL injection and XSS vulnerabilities by generating sample inputs, symbolically tracking taint information through execution (including through database accesses), and automatically generating concrete exploits. The typical drawbacks of such dynamic approaches are the limited test coverage and the missing ability to crawl a given site "deep" enough. This insight is confirmed by Doupé et al., who tested eleven black-box dynamic vulnerability scanners and found that whole classes of vulnerabilities are not well-understood and cannot be detected by the state-of-the-art scanners [7].

Static Analysis We perform static analysis of PHP code and use the concept of *block summaries* as proposed by Xie and Aiken [30] and later on refined by Dahse and Holz [6]. Our analysis tool extends these ideas and we improved the modeling of the language. More precisely, we introduce more data symbols (e.g., to analyze array accesses in a more precise way) and enhance the analysis of built-in functions such that we can perform a taint analysis for persistent data stores. Furthermore, we optimized the inter-procedural analysis to refine our string analysis results. This enables us to analyze the *two* distinct data flows that lead to second-order vulnerabilities: (i) source to PDS and (ii) PDS to sink. As a result, we are able to detect vulnerabilities missed by these approaches. Pixy [11] and Saner [2] are other static code analysis tools for web applications, but both do not recognize second-order vulnerabilities.

There are static analysis approaches that target other classes of security vulnerabilities. For example, SaferPHP [25] attempts to find *semantic attacks* (e.g., denial of service attacks due to infinite loops caused by malicious inputs, or unauthorized database operations due to missing security checks) within web applications. RoleCast [24] identifies security-critical variables and applies role-specific variable consistency analysis to identify missing security checks, while Phantm [17] detects type errors in PHP code. Such kinds of software defects are out of scope for our analysis.

Static Second-Order Analysis The work closest related to our approach is MiMoSA [3]. It is an extension of Pixy [11] to detect multi-module data flow and work flow vulnerabilities. The data flow through databases is modeled, however, it uses a dynamic approach for the reconstruction of SQL queries. Moreover, it focuses on the detection of the work flow of an application and does not handle neither other types of PDS nor multi-step exploits. In comparison, only three data flow vulnerabilities were detected in *Scarf*, whereas our approach detected 37 second-order vulnerabilities and one multi-step exploit.

Zheng and Zhang proposed an approach to detect *atomicity violations* in web applications regarding external resources [31], which can be seen as being closely related to second-order vulnerabilities since such concurrency errors are a pre-condition for second-order exploits. They perform a context- and path-sensitive interprocedural static analysis to automatically detect atomicity violations on shared external resources. The tools *NewsPro* and *Scarf* are included into their evaluation, but the authors did not find any of the second-order vulnerabilities detected by our approach. As such, our approach outperformed prior work on static detection of second-order vulnerabilities.

6 Conclusion and Future Work

In this paper, we demonstrated that it is possible to statically model the data flow through persistent data stores by collecting all storage writings and readings. At the end of the analysis, we can determine if data read from a persistent store can be controlled by an attacker and if this leads to a security vulnerability. Our prototype implementation demonstrated that this is an overlooked problem in practice: we identified more than 150 vulnerabilities in six popular web applications and showed that prior work in this area did not detect these software defects. From a broader perspective, our approach can be broken down to the problem of statically reconstructing all strings that can be generated at runtime by the application and thus, is limited by the halting problem.

Future work includes modeling the data flow when prepared statements are used, supporting more SQL features, and analyzing data flow through file content. Also, path-sensitive sanitization and aliasing should be analyzed more precisely [32].

References

[1] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. D. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Softw. Eng. 36*, 4 (2010).

[2] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy* (2008).

[3] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-Module Vulnerability Analysis of Web-based Applications. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[4] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy* (2010).

[5] BOJINOV, H., BURSZTEIN, E., AND BONEH, D. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[6] DAHSE, J., AND HOLZ, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[7] DOUPÉ, A., COVA, M., AND VIGNA, G. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2010).

[8] GUNDY, M. V., AND CHEN, H. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[9] HALFOND, W. G., VIEGAS, J., AND ORSO, A. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (2006).

[10] HILLS, M., KLINT, P., VINJU, J., AND HILLS, M. An Empirical Study of PHP Feature Usage. In *International Symposium on Software Testing and Analysis (ISSTA)* (2013).

[11] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications. *Journal of Computer Security 18*, 5 (08 2010).

[12] KERNIGHAN, B. W., AND PIKE, R. The Practice of Programming. In *Addison-Wesley, Inc* (1999).

[13] KHOURY, N., ZAVARSKY, P., LINDSKOG, D., AND RUHL, R. Testing and Assessing Web Vulnerability Scanners for Persistent SQL Injection Attacks. In *Proceedings of the First International Workshop on Security and Privacy Preserving in e-Societies* (2011), SeceS '11, pp. 12–18.

[14] KIEYZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *International Conference on Software Engineering (ICSE)* (2009).

[15] KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *ACM Symposium On Applied Computing (SAC)* (2006).

[16] KLEIN, A. Cross-Site Scripting Explained. *Sanctum White Paper* (2002).

[17] KNEUSS, E., SUTER, P., AND KUNCAK, V. Phantm: PHP Analyzer for Type Mismatch. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2010).

[18] LIVSHITS, B., AND CUI, W. Spectator: Detection and Containment of JavaScript Worms. In *USENIX Annual Technical Conference* (2008).

[19] MCALLISTER, S., KIRDA, E., AND KRUEGEL, C. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[20] MICROSOFT DEVELOPER NETWORK LIBRARY. Naming Files, Paths, and Namespaces. `http://msdn.microsoft.com/en-us/library/aa365247(VS.85)`, as of February 2014.

[21] NADJI, Y., SAXENA, P., AND SONG, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[22] SCHOLTE, T., ROBERTSON, W., BALZAROTTI, D., AND KIRDA, E. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *ACM Symposium On Applied Computing (SAC)* (2012).

[23] SEKAR, R. An Efficient Black-Box Technique for Defeating Web Application Attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[24] SON, S., MCKINLEY, K. S., AND SHMATIKOV, V. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2011).

[25] SON, S., AND SHMATIKOV, V. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)* (2011).

[26] SUN, F., XU, L., AND SU, Z. Client-side Detection of XSS Worms by Monitoring Payload Propagation. In *European Symposium on Research in Computer Security (ESORICS)* (2009).

[27] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRÜGEL, C., AND VIGNA, G. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2007).

[28] W3TECHS. World Wide Web Technology Surveys. `http://w3techs.com/`, as of February 2014.

[29] WASSERMAN, G., AND SU, Z. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE)* (2008).

[30] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium* (2006).

[31] ZHENG, Y., AND ZHANG, X. Static Detection of Resource Contention Problems in Server-side Scripts. In *International Conference on Software Engineering (ICSE)* (2012), pp. 584–594.

[32] ZHENG, Y., ZHANG, X., AND GANESH, V. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, pp. 114–124.

ASM: A Programmable Interface for Extending Android Security

Stephan Heuser*
stephan.heuser@trust.cased.de
Intel CRI-SC at TU Darmstadt

Adwait Nadkarni*
apnadkar@ncsu.edu
North Carolina State University

William Enck
enck@cs.ncsu.edu
North Carolina State University

Ahmad-Reza Sadeghi
ahmad.sadeghi@trust.cased.de
TU Darmstadt / CASED

Abstract

Android, iOS, and Windows 8 are changing the application architecture of consumer operating systems. These new architectures required OS designers to rethink security and access control. While the new security architectures improve on traditional desktop and server OS designs, they lack sufficient protection semantics for different classes of OS customers (e.g., consumer, enterprise, and government). The Android OS in particular has seen over a dozen research proposals for security enhancements. This paper seeks to promote OS security extensibility in the Android OS. We propose the Android Security Modules (ASM) framework, which provides a *programmable interface* for defining new reference monitors for Android. We drive the ASM design by studying the authorization hook requirements of recent security enhancement proposals and identify that new OSes such as Android require new types of authorization hooks (e.g., replacing data). We describe the design and implementation of ASM and demonstrate its utility by developing reference monitors called *ASM apps*. Finally, ASM is not only beneficial for security researchers. If adopted by Google, we envision ASM enabling in-the-field security enhancement of Android devices without requiring root access, a significant limitation of existing bring-your-own-device solutions.

1 Introduction

Consumer operating systems are changing. Android, iOS, and Windows 8 place a high priority on the user-application experience. They provide new abstractions for developing user-applications: applications fill the screen; they have complex lifecycles that respond to user and system events; and they use semantically rich OS provided application programming interfaces (APIs) such as "get location," "take picture," and "search address book." The availability of these semantically rich OS APIs vastly simplifies application development, and has led to an explosive growth in the number and diversity of available applications.

These functional changes caused OS designers to rethink security. The new application abstractions both enable and necessitate assigning each user application to a unique protection domain, rather than executing all user applications with the user's ambient authority (the norm in traditional OSes such as Windows and UNIX). By default, each application's protection domain is small, often containing only the OS APIs deemed not to be security sensitive and the files it creates. The application must be granted capabilities to access the full set of semantically rich OS APIs. This security model provides a better approximation of least privilege, which limits both the impact of an exploited application, as well as the authority granted to a Trojan. However, how and when to grant these privileges has been the topic of much debate [16].

For the last several years, the security research community has contributed significant discourse on the right security architecture for these new operating systems. Android has been the focus of this discourse, mostly due to its open source foundation, widespread popularity for mobile devices, and the emergence of malware targeting it. In the relatively short period of time since the Android platform's initial release in 2008, there have been more than a dozen proposals for new Android security architectures [15, 24, 14, 23, 10, 7, 8, 37, 6, 19, 18, 12, 9, 22, 29]. As we discuss in this paper, while these security architecture proposals have very diverse motivations, their implementations often share hook placements and enforcement mechanisms.

The primary goal of this paper is to promote OS security extensibility [33] in the Android platform. History has shown that simply providing type enforcement, information flow control, or capabilities does not meet the demands of all potential OS customers (e.g., consumers,

*These authors contributed equally to this work.

enterprise, government). Therefore, an extensible OS security interface must be *programmable* [33]. In short, we seek to accomplish for Android what the LSM [34] and TrustedBSD [32] frameworks have provided for Linux and BSD, respectively. What makes this task interesting and meaningful to the research community is the process of determining the correct semantics of authorization hooks for this new OS architecture.

In this paper, we propose the *Android Security Modules* (ASM) framework, which provides a set of authorization hooks to build reference monitors for Android security. We survey over a dozen recent Android security architecture proposals to identify the hook semantics required of ASM. Of particular note, we identify the need to (1) replace data values in OS APIs, and (2) allow third-party applications to define new ASM hooks. We design and implement an open source version of ASM within Android version 4.4 and empirically demonstrate negligible overhead when no security module is loaded. ASM fulfills a strong need in the research community. It provides researchers a standardized interface for security architectures and will potentially lead to field enhancement of devices without modifying the system firmware (e.g., BYOD), if adopted by Google.

This paper makes the following contributions:

- *We identify the authorization hook semantics required for new operating systems such as Android.* The Android OS is responsible for enforcing more than just UNIX system calls. Android includes semantically rich OS APIs and new application lifecycle abstractions that must be included in OS access control. We also identify the need for authorization hooks to replace data values and for third-party applications to introduce new authorization hooks.

- *We design and implement the extensible Android Security Modules (ASM) framework.* ASM brings OS security extensibility to Android. It allows multiple simultaneous *ASM apps* to enforce security requirements while minimizing performance overhead based on the required authorization hooks.

- *We implement two example ASM apps to demonstrate the utility of the ASM framework.* ASM allowed the fast development of useful example ASM apps with functionalities similar to MockDroid [6] and password protected apps.

Finally, we envision multiple ways in which ASM can benefit the security community. ASM currently provides great value to researchers with the ability to modify the source code of a device. It provides a modular interface to define callbacks for a set of authorization hooks that provide mediation of important protection events. As the Android OS changes, only the ASM hook placements need to change, eliminating the need to port each research project to new versions. ASM can provide even greater benefit if it is adopted into the Android Open Source Project (AOSP): ASM apps can be added without source code modification. Ultimately, *we envision an interface that allows enterprise IT and researchers to load ASM apps on production phones without root access.*

The remainder of this paper proceeds as follows. Section 2 provides a short background on Android. Section 3 defines high level goals that underlie the ASM design. Section 4 surveys recent work enhancing Android security and identifies a common set of authorization hook semantics. Section 5 describes the ASM design. Section 6 evaluates the utility and performance of ASM. Section 7 highlights related work on OS security extensibility. Section 8 concludes.

2 Background

The Android OS is based on a Linux kernel, but provides a substantially different application abstraction than found in traditional Linux desktop and server distributions. Android applications are written in Java and compiled into a special DEX bytecode that executes in Android's Dalvik virtual machine. Applications may optionally contain native code components. Application functionality is divided into components. Android defines four types of components: *activity*, *service*, *broadcast receiver*, and *content provider*. The application's user interface is composed of a set of activity components. Service components act as daemons, providing background processing. Broadcast receiver components handle asynchronous messages. Content provider components are per-application data servers that are queried by other applications.

Application components communicate with one another using Binder interprocess communication (IPC). Binder provides message passing (called *parcels*) and thread management. In addition to data values, parcels can pass references to other binder objects as well as file descriptors. When an application holds a reference to a service component binder object, it can execute remote procedure calls (RPCs) for any methods defined by that service. Most of Android's semantically rich OS APIs are implemented as RPCs to OS defined service components. The OS also defines several content provider components (e.g., address book) that are queried using special RPC methods. It should be noted that while developers are encouraged to use Binder IPC, Android also supports standard Linux IPC mechanisms, for example domain sockets or pipes.

Applications often interface with Binder indirectly using *intent messages*. The intent message abstraction is used for communication between activity and broadcast

receiver components, as well as starting service components. Intent messages can be addressed to implicit *action strings* that are resolved by the Activity Manager Service (AMS). Intent messages and action strings allow end users and OEMs to customize the applications used to perform tasks. The AMS resolves the desired target application and component, starting a new process or thread if necessary.

Android enforces component security requirements using *permissions* (i.e., text strings that represent capabilities). Android defines a set of core permissions for protecting OS resources and applications, but third-party application developers can define new permissions that are enforced using the same mechanisms as OS permissions. Permissions are granted to applications on install and stored in the Package Manager Service (PMS). Android places authorization hooks (implemented as a family of *checkPermission()* methods) in the AMS as well as OS service component RPC methods. *checkPermission()* is called along with the process identifier (PID) of the caller and the appropriate permission string. Calling *checkPermission()* invokes an RPC in the PMS, which returns *granted* if the caller's PID belongs to an application that is granted the permission, and throws a security exception if it is denied. However, not all permissions are enforced using *checkPermission()*. Permissions that control access to low-level capabilities are mapped to Linux group identifiers (GIDs). Such capabilities include opening network sockets and accessing the SDcard storage. For these permissions, corresponding GIDs are assigned to applications at installation time, and the kernel provides enforcement.

3 Design Goals

A secure operating system requires a *reference monitor* [2]. Ideally, a reference monitor provides three guarantees: complete mediation, tamperpoofness, and verifiability. We seek to provide a foundation for building reference monitors in Android. As with LSM [34], the ASM only provides the reference monitor interface hooks upon which authorization modules are built. Furthermore, similar to the initial design of LSM, our ASM design manually places hooks throughout Android.

We seek to design a programmable interface for building new security enhancements to the Android platform. Our design is guided by the following goals.

G1 *Generic authorization expressibility.* We seek to provide the reference monitor interface hooks necessary to develop both prior and future security enhancements for Android. Not all authorization modules will use all hooks, and hooks may need to be placed at different levels to obtain sufficient enforcement semantics.

G2 *Ensure existing security guarantees.* Android provides sandboxing guarantees to application providers. Allowing third-parties to extend Android's security framework potentially breaks those guarantees. Therefore, ASM's reference monitor interface hooks should only make enforcement more restrictive (e.g., fewer permissions or less file system access). Note that by only allowing more restrictive enforcement, we lose expressibility (e.g., for capability models).

G3 *Protect kernel integrity.* As an explicit extension to Goal **G2**, we must maintain kernel integrity. Some authorization modules will require hooks within the Linux kernel. We cannot provide the LSM interface to third-parties without some controls. We explore several methods of exposing this functionality in Section 5.4.5.

G4 *Multiple authorization modules.* While there have been proposals for supporting multiple LSMs [27], official support for multiple authorization modules in Linux has not been adopted at the time of writing. We see benefit in allowing multiple ASM modules (e.g., personal and enterprise) and seek to design support for multiple authorization modules into the design of ASM. Achieving multiple authorization modules requires carefully designing the architecture to address potential conflicts.

G5 *Minimize resource overhead.* When no authorization module is loaded, ASM should have negligible impact on system resources (e.g., CPU performance, energy consumption). Furthermore, given the wide variety of authorization hook semantics, we recognize that not all authorization modules will require all hooks. Since some hooks have more overhead than others, we seek to design ASM such that different hooks can be enabled and disabled to minimize overhead.

Threat Model: ASM assumes that the base Android OS and services are trusted. That is, our trusted computing base (TCB) includes the Linux kernel, the AMS, the PMS, and all OS service and content provider components. We assume that third-party applications have complete control over their process address spaces. That is, any authorization hooks placed in framework code that executes within the third-party application's process is untrusted. Finally, since third-party applications can include their own authorization hooks, they must be trusted to mediate the protection events they define.

4 Authorization Hook Semantics

The underlying motivation of ASM is to provide a programmable interface to extend Android security. Re-

Table 1: Classification of authorization hook semantics required by Android security enhancements

System	Android ICC	Package Manager	Sensors / Phone Info	Fake Data	System Content Providers	File Access	Network Access	Third Party Extension
MockDroid [6]		✓	✓	✓	✓		✓	
XManDroid [7]	✓	✓	✓			✓	✓	
TrustDroid [8]	✓	✓			✓	✓	✓	
FlaskDroid [9]	✓	✓	✓	✓	✓	✓	✓	✓
CRePE [10]	✓		✓					
Quire [12]	✓	✓						
TaintDroid [14]	✓		✓			✓	✓	
Kirin [15]		✓						
IPC Inspection [18]	✓	✓						
AppFence [19]	✓	✓	✓	✓	✓	✓	✓	
Aquifer [22]	✓					✓	✓	
APEX [23]	✓	✓	✓					
Saint [24]	✓	✓						✓
SEAndroid [29]	✓	✓				✓	✓	
TISSA [37]			✓	✓	✓			

cently, Google adopted the UNIX-level portion of the SEAndroid [29] project into AOSP. However, Android security is significantly more complex than simply mediating UNIX system calls. Nearly all application communication occurs through Binder IPC, which from a UNIX perspective is an *ioctl* to `/dev/binder`. Mediating the higher level application communication has been the focus of most Android security research. The goal of this section is to explore these different proposals to identify a common set of authorization hooks semantics. That is, we seek to satisfy Goal **G1** by surveying existing proposals to enhance Android security.

Academic and industry researchers have proposed many different security enhancements to the Android OS. These enhancements have a wide range of motivations. For example, Kirin [15] places constraints on permissions of applications being installed. Frameworks such as Saint [24], XManDroid [7] and TrustDroid [8] focus on mediating communication between components in different applications. FlaskDroid [9] and the aforementioned SEAndroid [29] project also mediate component interaction as a part of their enforcement. Aquifer [22] enforces information flow control policies that follow the user's UI workflow. IPC Inspection [18] and Quire [12] track Android intent messages through a chain of applications to prevent privilege escalation attacks. TaintDroid [14] and AppFence [19] dynamically track privacy sensitive information as it is used within an application. APEX [23] and CRePE [10] provide fine-grained permissions. TISSA [37], MockDroid [6], and AppFence [19] allow fine-grained policies as well as allow the substitution of fake information into Android APIs. While these proposals have diverse motivations, many share authorization hook semantics.

Table 1 classifies this prior work by authorization hook semantics. Nearly all of the proposals modify Android's Activity Manager Service (AMS) to provide additional constraints on Inter-Component Communication (ICC). The Package Manager Service (PMS) is also frequently modified to customize application permissions. Permissions are also occasionally customized by modifying the interfaces to device sensors and system content providers containing privacy sensitive information (e.g., address book). Several proposals also require authorization hooks for file and network access, which are enforced in the Linux kernel.

The table also denotes two areas that are nonstandard for OS reference monitors. The first hook semantics is the use of fake data. That is, instead of simply allowing or denying a protected operation, the hook must modify the value that is returned. This third option is often essential to protecting user privacy while maintaining usability. For example, the geographic coordinates of the north pole, or maybe a coarse city coordinates can be substituted for the devices actual location. Replacing unique identifiers (e.g. IMEI or IMSI) to combat advertising tracking is a further example. The second interesting hook semantics is the inclusion of third-party hooks. That is, a third-party application wishes the OS reference monitor to help enforce its security goals.

Finally, TaintDroid [14] and AppFence [19] use fine-grained taint tracking. They modify Android's Dalvik environment to track information within a process. However, dynamic taint tracking has false negatives, which may lead to access control circumvention. It also incurs more performance overhead than may be tolerable for some environments. In this work, we only consider mediation at the process level. Therefore, TaintDroid and AppFence cannot be built on top of ASM. However, this does not preclude researchers from combining TaintDroid with ASM.

5 ASM Design

The authorization hooks identified in the previous section describe semantically what to mediate, but not how to mediate it. Existing Android security enhancements

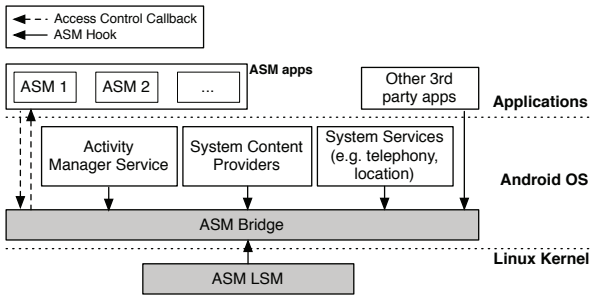


Figure 1: ASM framework architecture

define hooks in different ways, not all of which provide correct or complete mediation. ASM provides a reference monitor interface for building new reference monitors. By doing so, ASM allows reference monitor developers to focus on their novel security enhancements and not on placing hooks correctly. It also allows separate scrutiny of authorization hook placement that benefits all reference monitors built on top of ASM.

Figure 1 shows the ASM framework architecture. Reference monitors are implemented as *ASM apps*. Each ASM app registers for a unique set of authorization hooks, specifying a callback for each. When a protected operation occurs, ASM automatically invokes the callback in the ASM app. The ASM reference monitor interface is contained within the *ASM Bridge*. In addition to managing ASM apps, the ASM Bridge receives protection events from authorization hooks placed throughout the Android OS. Since Android places functionality in multiple userspace processes, authorization hooks only notify the ASM Bridge if the hook is explicitly enabled. ASM also supports authorization hooks within the Linux kernel. To achieve kernel authorization, a special ASM LSM performs upcalls to the ASM Bridge, again only doing so for hooks explicitly enabled.

This section details the design of the ASM framework. We use the following terminology. A *protection event* is an OS event requiring access control. *Authorization hooks* are placed throughout the Android OS, which invoke a callback in the ASM Bridge. The ASM Bridge defines *reference monitor interface hooks*, for which ASM apps register *hook callbacks*. Finally, we frequently refer to the ASM framework as a whole simply as *ASM*.

5.1 ASM Apps

Reference monitors are built as ASM apps. They are developed using the same conventions as other Android applications. The core part of an ASM app is a service component that implements the reference monitor hook interface provided by ASM. There are three main functionalities that must be provided within this service. Finally, the registration interface itself is protected by Android permissions.

ASM App Registration: An ASM app must register itself with the ASM Bridge after it is installed. The time of registration depends on logic in the specific ASM app. For example, the ASM app could register itself automatically after install, or it could provide a user interface to enable and disable it. When the ASM Bridge receives the registration, it updates its persistent configuration. To activate the ASM app, the device must reboot. We require a reboot to ensure ASM apps receive all protection events since boot, which may impact their protection state.

Hook Registration: The ASM app service component is started by ASM during the boot process. At this time, the ASM app registers for reference monitor interface hooks for which it wishes to receive callbacks. Different hooks incur different overheads. ASM only enables a reference monitor hook if it is registered by an ASM app. Therefore, ASM app developers should only register for the hooks required for complete mediation. Finally, if the ASM registers for hooks defined by a third-party application (Section 5.4.4), the application developer and the ASM app developer must agree on naming conventions.

Handling Hook Callbacks: Once an ASM app registers for a reference monitor interface hook, it will receive a callback whenever the corresponding protection event occurs. The information provided in the callback is hook-specific. The ASM app returns the access control decision to the ASM Bridge. As discussed in Section 5.3, some hooks allow the callback to replace data values. Finally, similar to registration for third-party hooks, the ASM app developer must coordinate with the application developer for information passed to the callback.

Registration Protection: Reference monitors are highly privileged. While ASM does not allow an ASM app to override existing Android security protections (Goal **G2**), ASM must still protect the ability to receive callbacks. ASM protects callbacks using Android's existing permission model. It defines two permissions: `REGISTER_ASM` and `REGISTER_ASM_MODIFY`. The ASM Bridge ensures that an ASM app has the `REGISTER_ASM` permission during both ASM app registration and hook registration. Finally, since replacing data values in an access control callback has greater security implications, the ASM Bridge ensures the ASM app has the `REGISTER_ASM_MODIFY` permission if it registers for a hook that allows data modification. This allows easy ASM app inspection to identify its abilities.

ASM App Deployment: How the ASM permissions are granted has a significant impact on the practical security of devices. Previous studies [17] have demonstrated that end users frequently do not read or understand Android's install time permissions. Therefore, malware may attempt to exploit user comprehension of permissions and gain ASM app privileges. To some extent, this threat is

mitigated by our goal to ensure existing security guarantees (Goal **G2**). Different ASM app deployment models can also mitigate malware. In the use case where researchers change AOSP source code, these permissions can be bound to the firmware signing key, thereby only allowing the researchers' ASM apps to be granted access. In the case where ASM is deployed on production devices, ASM could follow the security model used by device administration API. That is, a secure setting that is only modifiable by users would enable whether ASM apps can be used. An alternative is to use a model similar to Android's "Unknown sources" setting for installing applications. That is, unless a secure user setting is selected, only Google certified ASM apps can be installed.

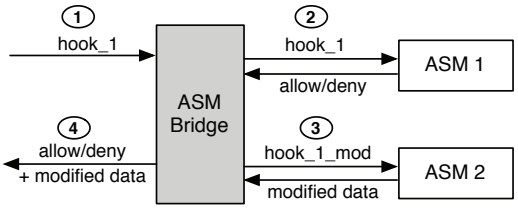
5.2 ASM Bridge

The ASM Bridge 1) provides the reference monitor interface, and 2) coordinates protection events that occur in authorization hooks placed throughout the Android OS, as well as third-party applications. As discussed in Section 5.1, ASM apps notify the ASM Bridge of their existence via an ASM app registration followed by individual hook registrations. We now discuss several reference monitor interface considerations.

Per-Hook Activation: All reference monitor interface hooks are deactivated by default. Each authorization hook maintains an activation state variable that determines whether or not the ASM Bridge is notified of protection events. This approach eliminates unnecessary IPC and therefore improves performance (Goal **G5**) when no ASM app requires a specific hook. Likewise, this approach allows ASM to achieve negligible overheard when no ASM apps are loaded (see Section 6.2).

When an ASM app registers a callback for a deactivated hook, the ASM Bridge activates the hook by notifying the corresponding authorization hook implementation. ASM maintains a list of active hooks in each OS component (e.g., OS service component, OS content provider component). When a protection event occurs, the OS component creates an access control bundle that is sent to the ASM Bridge. When the ASM Bridge receives the access control bundle for a hook, it is forwarded to each ASM app that registered for the hook. Similarly, the ASM LSM in the kernel (Section 5.4.5) maintains a separate activation state variable per hook and performs an upcall for each protection event.

Callback Timeouts: The ASM Bridge is notified of protection events via synchronous communication. Authorization hooks in userspace communicate with the ASM Bridge using Binder IPC, and the ASM LSM uses synchronous upcalls, as described in Section 5.4.5. The ASM Bridge then uses synchronous Binder IPC to invoke all ASM app callbacks for the hook corresponding



1. ASM Bridge receives a callback.
2. The normal hook is invoked.
3. The modified data hook (i.e. mod) is invoked.
4. ASM Bridge returns the result for the initial callback.

Figure 2: ASM Hook Invocation

to the protection event. If the ASM app callback implementation is buggy, the authorization hook may stall execution. Therefore, ASM has the ability to set timeouts on callback execution. If a timeout occurs, the ASM Bridge conservatively assumes access is denied.

Master Policy: ASM supports multiple simultaneous ASM apps (Goal **G4**). This goal is motivated by multi-stakeholder scenarios, e.g. users, administrators, and device manufacturers installing ASM apps on a device. When more than one ASM app is active, a reconciliation strategy is required to handle potential conflicts between access control decisions. The correct conflict resolution strategy is highly use-case specific. Therefore, providing a general solution is infeasible [9].

ASM addresses this problem using a master policy that defines policy conflict reconciliation. For our implementation and evaluation, we use a consensus strategy. That is, all active ASM apps must grant an access control decision for the action to be allowed. Similar to FlaskDroid [9], the master policy can be easily modified to support other conflict resolution strategies [26, 21]. For example, a priority-based resolution policy hierarchically orders ASM apps, and a voting policy allows an action if a specified threshold of ASM apps grant it.

5.3 Callbacks Modifying Data

Before discussing the reference monitor interface hooks provided by ASM, we must describe one last concept. While most ASM apps require a simple allow/deny access control interface, some may benefit from the ability to modify data values. For example, MockDroid [6] modifies values (e.g., IMEI, location) returned by OS APIs before they are sent to applications. ASM supports data modifications by providing a special hook type.

Each reference monitor interface hook that potentially requires data replacement is split into two variants: 1) *normal*, which allows the corresponding callback to simply allow or deny the event, and 2) *modify*, which allows the corresponding callback to modify the value returned by the OS API or content provider, in addition to specifying allow or deny. As mentioned in Section 5.1, modifying data has a greater security sensitivity,

```
1 // Callback received by the ASM Bridge:
2 int start_act(inout Intent intent, in String
   resolvedType, in ActivityInfo activity, int
   requestCode, int callingPid, int callingUid);
3 // Callback to individual ASMs (No modify data):
4 int start_act(in Intent intent, in String
   resolvedType, in ActivityInfo activity, int
   requestCode, int callingPid, int callingUid);
5 // Callback to individual ASMs (Modify data):
6 int start_act_mod(in Intent intent, inout Bundle
   extras, in String resolvedType, in ActivityInfo
   activity, int requestCode, int callingPid, int
   callingUid);
```

Listing 1: Example Callback Prototypes Modifying Data

and therefore registration of a *modify* callback requires the `REGISTER_ASM_MODIFY` permission.

Figure 2 shows how the ASM Bridge manages normal and modify hooks. To reduce the overhead of handling authorization hooks, the ASM Bridge is only notified once per protection event. The ASM Bridge then manages the normal and modify versions, returning the access control decision and modified data value (if needed) to the authorization hook. Additionally, the ASM Bridge invokes all of the normal callbacks before the modify versions. This approach allows a performance improvement if a consensus master policy is used (Section 5.2). In this case, if a normal hook denies access, the modify callbacks do not need to be called.

Example 1: Listing 1 explains this distinction further via example. The listing shows the callback prototypes for the `start_activity` protection event. The first prototype shown, `start_act()`, is the ASM Bridge callback used by the authorization hook in the Activity Stack subsystem of Android's AMS. This hook is invoked after intent resolution but before the chosen activity component is started. The hook includes 1) the intent message from the caller, 2) information about the activity to be started, 3) the caller's identity, and 4) additional information for the current event. By marking *intent* as *inout* (a directive defined in the *Android Interface Definition Language*), the ASM Bridge can modify it.

The ASM Bridge splits `start_act()` into the normal and modify versions. To ensure restrictive enforcement, ASM apps can modify only the *extras* field supplied by the caller. It cannot modify information that has been reviewed by the user or the OS, such as the action string or the target activity. To ensure this restriction, the ASM Bridge makes the intent immutable, but supplies a mutable *Bundle* of extras extracted from the intent to the ASMs registered for the modify data hook. The modified extras received by the ASM Bridge are then set back to the intent before the initial callback from the Activity Stack to the ASM Bridge returns.

```
1 // Callback received by the ASM Bridge:
2 int resolveActivity_mod(inout List<ResolveInfo>
   resolvedList, in String resolvedtype, int userId,
   inout Intent intent, int callingPid, int callingUid);
3 // Callback to individual ASM apps (Modify data):
4 int resolveActivity_mod(inout List<ResolveInfo>
   resolvedList, in String resolvedtype, int userId, in
   Intent intent, int callingPid, int callingUid, inout
   Bundle extras);
```

Listing 2: Resolve Activity Hook

5.4 Hook Types

ASM provides a reference monitor interface for authorization hooks placed throughout the Android OS. We now describe five general categories of hooks: 1) lifecycle hooks, 2) OS service hooks, 3) OS content provider hooks, 4) third-party app hooks, and 5) LSM hooks.

5.4.1 Lifecycle Hooks

ASM provides reference monitor hooks for component lifecycle events in the Activity Manager Service, the AMS subsystems, and the Package Manager Service. Hooks in this category include: resolving intents, starting activities and services, binding to services, dynamic registration of broadcast receivers, and receiving broadcast intents. We demonstrate the lifecycle hook category with the following example. Note that Example 1 is also a lifecycle hook.

Example 2: The `resolve_activity` protection event occurs within the Package Manager Service. The ASM authorization hook for `resolve_activity` is placed in the PMS after the intent has been resolved by the OS, but before a chooser with the resolved activities is presented to the user. This hook is motivated by systems such as Saint [24] and Aquifer [22], which refine the list of resolved applications based on access control policies. Note that refining the chooser list requires data modification, and therefore, `resolve_activity` is one of few hooks that only provide a modify version.

Listing 2 shows the callback prototypes defined for `resolve_activity`. The callback received by the ASM Bridge from the Android OS contains the list of resolved components. The ASM Bridge then executes an RPC to the ASM app callbacks registered for this hook. The RPC provides a modifiable resolved component list and Bundle extras. The other parameters are immutable. It is important to prevent the ASM from adding new apps to the list, thereby overriding the OS's restrictions (Goal **G2**). Therefore, we compute the set intersection of the original list and the modified list, and return the result to the authorization hook. When multiple ASM apps register for this hook, the ASM Bridge calls the hook callback for each ASM app, providing the modified data from the previous invocation as input.

5.4.2 OS Service Hooks

Lifecycle hooks include mediation for inter-component communication using intent messages. However, ASM apps also require mediation for OS APIs providing functionality such as getting the geographic location and taking pictures. Android implements this functionality in different service components designated as *system services*, e.g., location and telephony services.

ASM uses Android's AppOps subsystem to place the authorization hooks for many OS service hooks. AppOps is a very recent addition to AOSP. While there have been several popular media stories of hobbyist developers using AppOps to control per-application permissions, AppOps remains largely undocumented and is not yet available for public use. Based on our code inspection, AppOps appears to be an effort by Google to provide more flexible control of permission related events. Conceptually, AppOps is an Android security enhancement and could be implemented as an ASM app. We discuss AppOps as an ASM app further in Section 7.

The ASM authorization hooks for services use the AppOps syntax. AppOps defines opcodes for different operations, e.g., `OP_READ_CONTACTS` or `OP_SEND_SMS`. To identify the application performing an operation, the Linux uid and the package name of the application are used. ASM uses a single authorization hook in AppOps to call the ASM Bridge. The ASM Bridge decodes the opcode and translates it into an ASM hook.

AppOps supports graceful enforcement. That is, it returns empty data instead of throwing a Security Exception wherever possible (e.g., in Cursors). As a result, apps do not crash when they are denied access to resources. On the other hand, AppOps does not allow data values to be modified at runtime. Therefore, ASM adds specific data modification hooks. We also needed to extend AppOps with several hooks for privacy sensitive operations (e.g., `getDeviceId()`, `onLocationChanged()`). We now discuss two examples, including both regular AppOps hooks and ASM's data modification hooks.

Example 3: Listing 3 shows the callback prototype for the AppOps hook for sending an SMS (`OP_SEND_SMS`). The ASM Bridge receives the generic `appOpsQuery()` callback and translates the opcode to the `sendSms()` hook. ASM apps registered for the `sendSms()` hook receive a callback whenever an SMS message is sent.

Example 4: Listing 4 shows the data modification callback prototype for the `getDeviceId()` OS API call in the PhoneSubInfo (i.e., telephony) service. The ASM Bridge receives a callback from the authorization hook and executes the `getDeviceId_mod()` callback in ASM apps. ASM apps receiving this callback can return *deny* or *allow*. If the return value is *allow*, the ASM app can also place a custom value in the first index of the `device_ids` array. This value will be sent to the Android application that invoked `getDeviceId()`, instead of the real device ID.

```
1 // Callback received by the ASM Bridge:
2 int appOpsQuery(int opcode, int callingUid, String
   packageName);
3 // Here, opcode = OP_SEND_SMS
4 // Callback to individual ASMs:
5 int sendSms(int callingUid, String packageName);
```

Listing 3: AppOps Hook for Sending SMS

```
1 // Callback received by the ASM Bridge:
2 int getDeviceId(int callingUid, out String[]
   device_ids);
3 // Callback to individual ASMs (Modify data):
4 int getDeviceId_mod(int callingUid, out String[]
   device_ids);
```

Listing 4: getDeviceId Hook

5.4.3 Content Provider Hooks

Content provider components are daemons that provide a relational database interface for sharing information with other applications. The ASM Bridge receives callbacks from the OS content provider components (e.g., Calendar, Contacts, and Telephony). Separate hooks are required for the insert, update, delete and query functions. Authorization hooks for insert, update and delete must be invoked *before* the action is performed, to preserve the integrity of the provider's data. In contrast, the query function's hook is invoked *after* the execution, to allow filtering of the returned data.

The content provider query RPC returns a database Cursor object. The Cursor object not a *parcelable* type, and therefore the entire query response is not returned to the caller in a single Binder message. Therefore, ASM apps cannot filter the query. To account for this, we extract the Cursor contents into a parcelable *ASMCursor* wrapper around a *CursorWindow* object to include in the callback to the ASM Bridge.

The following example demonstrates the query interface. ASM only provides normal (i.e., no data modification) hooks for insert, delete, and update.

Example 5: Listing 5 shows the callback prototypes for the `CallLogProvider` OS content provider. The ASM Bridge receives the original query and the result wrapped in an ASMCursor. The callback is split into normal and modify hook variants. ASM apps that register for the normal hook get read access to the query and the result. ASM apps registered for the data modify hook can also modify the ASMCursor object. Both the hooks return allow and deny decisions via the return value.

Finally, we note that this use of a CursorWindow object to copy the entire content provider query response

```
// Callback received by the ASM Bridge:
int callLogQuery(inout ASMCursor cursor, in Uri uri,
 in String[] projection, in String selection, in
 String[] selectionArgs, in String sortOrder, int
 callingUid, int callingPid);
// Callback to individual ASMs (No modify data):
int callLogQuery(in ASMCursor cursor, in Uri uri, in
 String[] projection, in String selection, in String[]
  selectionArgs, in String sortOrder, int callingUid,
 int callingPid);
// Callback to individual ASMs (Modify data):
int callLogQuery_mod(inout ASMCursor cTemp, in Uri
 uri, in String[] projection, in String selection, in
 String[] selectionArgs, in String sortOrder, int
 callingUid, int callingPid);
```

Listing 5: CallLogProvider query hook

```
// Callbacks received by the ASM Bridge:
int hook_handler(in String name, in Bundle b);
int hook_handler_mod(in String name, inout Bundle b);
// Callback to individual ASMs (No modify data):
int hook_handler(in String name, in Bundle b);
// Callback to individual ASMs (Modify data):
int hook_handler_mod(in String name, inout Bundle b);
```

Listing 6: Third Party Hooks

into the ASM hook may lead to additional overhead when query responses are large. This is because Android uses a lazy retrieval of Cursor contents, only transferring portions of the response over Binder IPC as needed. One way to improve ASM query performance is to intercept the actual data access via Binder to modify data, rather than serializing the entire response. However, this will increase the number of callbacks to ASM apps, resulting in a trade-off. We will explore this and other methods of performance improvement in future work.

5.4.4 Third Party Hooks

ASM allows third-party Android applications to dynamically add hooks to the ASM Bridge. These hooks are valuable for extending enforcement into Google and device manufacturer applications (which are not in AOSP), as well as third-party applications downloaded from application markets (e.g., Google Play Store). Third-party hooks are identified by 1) a hook name, and 2) the package name of the application implementing the authorization hook. The complete hook name is a character string of the format `package_name:hook_name`. This naming convention provides third parties with their own namespaces for hooks. Note that third parties do not specify their package name; ASM obtains it using the registering application's uid received from Binder.

To receive callbacks for third-party hooks, ASM apps implement two generic third-party hook methods, shown in Listing 6. One method handles normal hook callbacks; the other method handles data modification hook callbacks. When the third-party application's authorization hook calls the ASM Bridge callback, it passes a generic Bundle object. The ASM forwards the Bundle to registered ASM apps for access control decisions. As with other ASM authorization hooks, third-party hooks are only activated when an ASM app registers for it.

ASM apps receive hook callbacks for all of their registered third-party hooks via a single interface (technically two callbacks, as in Listing 6). Within this callback, ASM apps must identify the third-party hook by name and must interpret the data in the Bundle based on the third-party application's specification. We assume that ASM apps that register for third-party hooks are aware of the absolute hook name and the contained attributes. The ASM app returns allow, deny, or allow along with a modification of the Bundle (for data modification hooks).

Finally, the third-party application developer must implement a special service component to receive hook activation and deactivation callbacks from the ASM Bridge. The ASM Bridge sends messages to this service to update the status of a hook. Third-party application developers must follow the message codes exposed by ASM for proper hook management.

5.4.5 LSM Hooks

ASM apps sometimes require mediation of UNIX-level objects such as files and network sockets. ASM cannot define authorization hooks for such objects in the userspace portion of the Android OS. Instead, authorization hooks must be placed in the Linux kernel. Fortunately, the Linux kernel already has the LSM reference monitor interface for defining kernel reference monitors. For example, `file_permission` and `socket_connect` LSM hooks mediate file and network socket operations, respectively.

The main consideration for ASM is how to allow ASM apps to interface with these LSM hooks. Several potential approaches exist. First, ASM could allow ASM apps to load LSM kernel modules directly. This approach is appropriate when the ASM app developer also has the ability to rebuild the device firmware. For example, one target audience for ASM is security researchers prototyping new reference monitors. In this case, the ASM app developer can create userspace and kernel components and provide communication between the two.

However, we would like to also allow ASM apps to mediate kernel-level objects without rebuilding the device firmware. Therefore, a second option is to develop a small mediation programming language that is interpreted by an ASM LSM. In this model, the ASM app developer programs access control logic within the interpreted language, and the logic is loaded along with the ASM on boot. Using an interpreted language would ensure kernel integrity (Goal **G3**).

Our current implementation uses a third option. We define a special ASM LSM that implements LSM hooks and performs synchronous upcalls to the ASM Bridge to complete the access control decision. Consistent with the

rest of the ASM design, the upcall is only activated when an ASM app registers for the corresponding reference monitor hook. To integrate our ASM LSM into the kernel without removing SEAndroid (Goal **G2**), we used an unofficial multi-LSM patch [27]. We implemented authorization hooks for many commonly used LSM hooks, including `file_permission` and `socket_connect`.

While the upcall approach initially sounds like it would have very slow performance, our key observation is that many ASM apps will require very few, if any, LSM hooks. For example, an ASM app for Aquifer [22] would only require the `file_permission` and `socket_connect` LSM hooks. Section 6.2 shows that both of the aforementioned hooks can be evaluated in userspace with reasonable performance overhead. Furthermore, placing all ASM app logic in one place (i.e., userspace) simplifies reference monitor design.

To improve access performance for large files, we implemented a cache with an expiration policy, where file accesses (euid, pid, inode, access_mask) and decisions received from ASM apps on those accesses are cached; and are invalidated if the accesses do not repeat within a timeout period of 1 ms. Since we cache and match the file inode as well as the accessing subject's effective uid and pid, we do not provide an attacker the opportunity of taking advantage of a race condition (i.e., requesting for a file less than 1ms after its access is granted).

Note that this approach may lead to a case where file access control is too coarse grained for a particular ASM app. For example, consider a situation where an application on the device reads a file continuously. An ASM app grants this application access, but if at some point during these accesses it wants to deny the access to this file, the `file_permission` hook is not triggered since the file is read before the timeout expires resulting in cache hits. To address this problem, we allow ASM apps to set this timeout. If multiple ASM apps set a timeout, the master policy can determine the timeout, e.g., the smallest timeout. ASM apps may also disable the cache, which provides all file access control callbacks to the ASM, but also degrades the performance of file reads.

5.5 ASM LSM

Finally, the ASM LSM provides two security features in addition to the LSM hook upcalls. First, it implements the `task_kill` LSM hook to prevent registered ASM apps from being killed. As we discuss in Section 6.1, some existing security enhancements can be disabled by killing their processes. Second, it implements the `inode_*xattr` LSM hooks to provide ASM apps access to their own unique extended attribute namespaces. That is, an ASM app can use file xattrs with a prefix matching its package name. No other applications can access these xattrs. File xattrs are needed by security enhancements such as Aquifer [22].

6 Evaluation

We evaluate the ASM framework in two ways. First, we evaluate the utility of ASM via case study by implementing two recent security enhancements for Android as ASM apps. Second, we evaluate the resource impact of ASM with respect to both performance and energy consumption. We implemented ASM on Android version 4.4.1, hence we use the Android 4.4.1 AOSP build as our baseline. All the experiments were performed on an LG Nexus 4 (GSM). The source code for ASM is available at `http://androidsecuritymodules.org`.

6.1 Case Studies

In this section, we evaluate the utility of ASM by implementing existing security solutions as ASM apps. We implement and study two examples: 1) MockDroid [6] and 2) AppLock [13]. Finally, we conclude this section with a summary of lessons learned.

6.1.1 MockDroid

MockDroid [6] is a system-centric security extension for the Android OS that allows users to gracefully revoke the privileges requested by an application without the app crashing. To do so, MockDroid provides a graphical user interface that allows the user to decide whether individual applications are presented real or fake responses when accessing sensitive system components.

Original Implementation: MockDroid extends Android's permissions model for accessing sensitive services by providing alternative "mock" versions. When users install an application, they choose to use the real or mock version of permissions. However, users can also revise this decision later using a graphical user interface. MockDroid stores the mapping between applications and permissions in an extension to Android's Package Manager Service. This policy store is the primary policy decision point in MockDroid.

MockDroid places enforcement logic in relevant Android OS components, as well as the kernel. If an application is assigned a mock permission, the Android OS component will return fake information. For example, if an application attempts to get the device IMEI, and it is assigned the mock version of `READ_PHONE_STATE`, then the telephony service will return a fake static IMEI instead of the device's real IMEI.

MockDroid also modifies the Linux kernel with enforcement logic. Recall from Section 2 that some permissions are enforced in the Linux kernel based on GIDs assigned to applications. MockDroid defines additional

Table 2: Hooks registered by the MockDroidASM app

Access to Fake	ASM Hook	ASM Callback
IMEI	device_id_mod	int getDeviceId_mod(String fake_imei[])
Fine/Coarse Location update	on_location_changed_mod	int onLocationChanged_mod(int uid, Location loc)
Internet Connection	socket_connect	int socket_connect(String family, String type, int uid)
Contacts Query	contacts_query_mod	int query_contacts_mod (ASMCursor c, String projection, ...)
Contacts Insert	contacts_insert	int contactsInsert(Uri uri, ContentValues values)
Contacts Delete	contacts_delete	int contactsDelete(Uri uri, String selection, String selectionArgs[], ...)
Contacts Update	contacts_update	int contactsUpdate(Uri uri, ContentValues values, String selection, ...)
Receive Broadcast	resolve_broadcast_mod	int resolveBroadcastReceivers_mod(List resolvedList, String resolvedtype, ..)

GIDs for mock permissions enforced by GID. For example, if the user selects to assign the mock version of the INTERNET permission to an application, it is assigned to the mock_inet group instead of the inet group. To enforce this mock permission, MockDroid modifies the inet runtime check in the Linux kernel (a check added by Android to Linux). In the modified check, if the application is in the mock_inet group, a socket timeout error is returned, simulating an unavailable network server.

MockDroidASM: We implemented an ASM app version of MockDroid called MockDroidASM. In addition to ASM permissions for hook registration, MockDroidASM must register for the PACKAGE_INSTALL hook to receive the package name and the list of requested permissions when each new application is installed. A MockDroidASM GUI also allows the user to configure which permissions to gracefully revoke from an application (e.g., INTERNET, READ_PHONE_STATE).

Instead of using additional mock permissions, MockDroidASM registers for the modify version of ASM hooks that are triggered when an application attempts to access sensitive system components. Since MockDroidASM needs to modify values returned to apps, it requests the REGISTER_ASM_MODIFY permission, as described in Section 5.3.

Table 2 shows the most important hooks used by MockDroidASM. For example, the device_id_mod hook allows MockDroidASM to fake the IMEI number of the device. On the kernel-level, MockDroidASM registers for the socket_connect hook to receive a callback when an application tries to connect to a network server. If INTERNET is revoked by the user, the MockDroidASM returns deny to the ASM LSM, which returns a socket timeout error to the application.

6.1.2 AppLock

AppLock [13] is an application available on the Google Play Store. It allows users to protect the user interface components of applications with a password. Users set a password to access the AppLock. They then selectively lock other third-party and system applications through AppLock's user interface. When the user tries to open a protected application, AppLock presents a password prompt, and the user must enter the correct password before the application can be used.

Original Implementation: AppLock requests install-time permissions for 1) getting the list of running apps, 2) overlaying its user interface over other applications, and 3) killing application processes. While AppLock does not require any modifications to Android's source code, it is uses energy very inefficiently. It can also be circumvented using an ADB shell (e.g., "am force-stop com.domobile.applock").

AppLock's *LockService* uses a busy loop to continuously query the Android operating system for the list of running applications while the screen is on. If the top application is protected by AppLock's policy, LockService overlays the current screen with a password prompt user interface. This interface stays on the screen, trapping all input until the correct password is entered. If the user decides to return from the lock screen without entering his password, AppLock kills the protected application. We have verified this execution via static analysis using ApkTool [1] as well as with another monitoring ASM app that registers for the start_service hook.

AppLockASM: We implemented an ASM app version of AppLock called AppLockASM. To provide the password-protected application functionality, AppLockASM simply registers for the start_activity hook. It then receives a callback whenever an activity component is started. When this occurs, AppLockASM displays its own lock screen. If the user enters the correct password, the start_activity event is allowed. If the user decides not to enter a password, it is denied. Unlike AppLock, AppLockASM never starts the target activity component without the correct password.

6.1.3 Summary

ASM considerably simplifies development of security modules such as AppLock and MockDroid. For example, the original AppLock app performs its functionality by starting a service in an infinite loop, a design that is inefficient in terms of power as well as latency. AppLockASM on the other hand needs to simply register for a callback with the ASM Framework. The AppLock implementation also prompts a lock screen after the app has already been started, and has to kill the app when the lock

Table 3: Performance - Unmodified AOSP, ASM with no reference monitor, and ASM with a reference monitor app

		ASM (ms)		Overhead (%)		Overhead (ms)	
Protection Event	**AOSP (ms)**	**w/o ASM app**	**w/ ASM app**	**w/o ASM app**	**w/ ASM app**	**w/o ASM app**	**w/ ASM app**
Start Activity	19.03±1.51	20.01±1.39	22.74±1.77	5.15	19.50	0.98	3.71
Start Service	3.89±0.31	4.6±0.41	8.42±0.61	18.25	116.45	0.71	4.53
Send Broadcast	2.18±0.24	4.48±0.69	6.45±0.55	105.50	196.71	2.30	4.27
Contacts Insert	121.41±5.98	120.48±5.25	135.39±6.35	-0.76	11.51	-0.93	13.98
Contacts Query	17.41±3.88	21.10±3.13	29.50±4.36	21.19	69.44	3.69	12.09
File Read	59.13±1.97	62.27±2.86	65.39±2.93	5.31	10.59	3.14	6.26
File Write	57.68±3.01	57.98±2.76	59.03±3.60	0.52	2.34	0.30	1.35
Socket Create	0.65±0.086	0.79±0.13	4.26±0.56	21.54	555.38	0.14	3.61
Socket Connect	1.61±0.21	1.65±0.22	5.13±0.32	2.48	218.63	0.04	3.52
Socket Bind	2.00±0.17	1.93±0.64	5.15±0.34	-3.5	157.50	-0.07	3.15

screen returns. This arbitrary killing of apps is prevented in the AppLockASM case, where the callback happens before the activity is started, and the activity starts only if the AppLockASM allows. This is also beneficial from the security point of view, as an AppLockASM-like app does not need to register for the permission to kill other apps, reducing the risk in case the locking app itself is malicious or malfunctions.

The original MockDroid implementation requires modifications to the Package Manager Service, and has to implement an entire parallel mock permission framework. This effort can be reduced by registering for a small number of ASM hooks, without having to modify system services.

A general lesson learned from these case studies is that the ASM architecture enables developers to easily implement complex system-centric security enhancements without the need for third party support. This broadens the outreach of ASM, and encourages third-party developers to engage in the development of sophisticated security solutions for Android-based devices.

6.2 Performance Overhead

To understand the performance implications of ASM, we micro benchmarked the most common ASM protection events for modules in Table 1. We performed each experiment 50 times in three execution environments: 1) AOSP, 2) ASM with no ASM app, and 3) ASM with one ASM app. The ASM app only registers for the callback of the tested protection event; all other callbacks remain deactivated. Since we are only interested in the performance overhead caused by framework, our test callback immediately returns *allow*. Table 3 shows the mean results with the 95% confidence intervals.

Lifecycle protection events: To test lifecycle protection events (i.e., start activity, start service, and send broadcast), we created an intent message and added a byte array as its data payload (i.e., extras Bundle). Each test type registered for the modify version of the ASM hook. We sent the intent for the respective type, pausing for five seconds between consecutive executions. Potential areas of overhead for using the hook include: 1) cost of establishing two additional IPCs, 2) marshalling and unmarshalling this data across the two IPCs, 3) ASM copying the extras Bundle when sending it to the ASM app, and 4) setting the returned Bundle back to the original intent. To estimate worst case performance, we chose a very large array (4KB) and registered our test ASM for modify data hooks. This worst case overhead, though relatively high, is not noticeable by the user due to its low absolute value. Additionally, most applications use files to share very large data values. We note that while send broadcast has a high overhead percentage, the wall clock overhead is in the order of milliseconds, which is negligible overhead for broadcasts.

Content provider protection events: Micro benchmarks for content providers were performed on the *Contacts Provider*. For this experiment, our ASM app registers for the `contacts_insert` callback. It proceeds to insert a new contact (first and last name) into the contacts database exposed by the ContactsProvider. The overhead observed is 11.51% and negligible in terms of its absolute value. We then registered for the `contacts_query_mod` hook, and performed a query on the same contact. Query has a greater overhead, which is attributable to marshalling/unmarshalling the data between the two IPC calls, and serialization of the Cursor object into a parcelable. A major cause of this overhead is also that the Content Provider Cursor is not populated when the query result is returned to the calling application, but is instead filled as and when the application uses it to retrieved values. As discussed in Section 5.4.3, future work will consider alternative methods of mediating query responses.

File access protection events: File micro benchmarks tested the `file_permission` hook, which uses an upcall from the kernel. To test file access performance, our test app performs an access (read/write) on a 5MB file. We pause for a second between successive executions. For writes, we do not see considerable overhead as the file is written in one shot to disk. Reads used a 16KB buffer and the default 1ms expiration time for caching access control decisions, as discussed Section 5.4.5.

Table 4: Energy overhead of ASM.

	Average Power Consumption (mW)	Overhead (%)
AOSP	670.42	-
ASM w/o ASM app	692.83	3.34
ASM w/ ASM app	732.98	9.33

Socket protection events: For socket operations, we tested the performance overhead for creating, binding and connecting to an IPv6 socket. Our test ASM app registered for the `socket_create`, `socket_bind`, and `socket_connect` callbacks. The absolute overhead is mainly caused by the callback to the userspace, and is a constant overhead for socket operations.

6.3 Energy Consumption

Energy consumption is a growing concern for mobile devices. To measure ASM's impact on energy consumption, we perform energy measurements in same three test environments as performance: 1) AOSP, 2) ASM with no ASM app, 3) ASM with one ASM app. The ASM app registers for all the hooks from the performance experiments. We use the Trepn profiler 4.1 [25] provided by Qualcomm to perform power measurements. Trepn uses an interface exposed by the Linux kernel to the power management IC used on the System on a Chip to measure energy consumption, a feature that is supported on a limited set of devices, including the LG Nexus 4. Trepn samples power consumption measurements every 100 ms. Average values are shown in the Table 4.

We monitor system energy consumption while running the test applications from Section 6.2. When the hooks are deactivated, we measured an energy consumption overhead of about 3.34%. Our ASM app used for the performance and energy consumption experiments measured an overhead of about 9.33%. This overhead is caused by the active authorization hooks in the relevant OS components and kernel, as well as the communication between the authorization hooks, the ASM Bridge, and the ASM app.

It should be noted that performing accurate energy consumption measurements on smartphones is a challenge. While we consider the individual measurements to be accurate, we acknowledge that the low sampling rate used by the Trepn profiler is problematic. However, each individual experiment is performed 50 times, therefore we believe Trepn's measurements to at least provide a rough estimate of the energy consumption overhead introduced by ASM.

7 Related Work

Section 4 discussed Android security enhancements that modify the Android firmware to achieve security mediation. As an alternative approach, Aurasium [35], AppGuard [5], RetroSkeleton [11] and Dr. Android and Mr. Hide [20] repackage applications with inline reference monitors (IRMs). While IRMs do not require firmware modification, rewriting frequently breaks applications, and the resulting mediation may be circumvented if API coverage is incomplete or native libraries are used. Placing access control mediation within the OS provides stronger guarantees.

ASM follows the methodology of the LSM [34] and TrustedBSD [32] reference monitor interface frameworks. Both frameworks have been highly successful. In Linux, LSM is widely used to extend Linux security enforcement. Version 3.13 of Linux kernel source includes SELinux [28], AppArmor [3], Tomoyo [31], Smack [30], and Yama [36] LSMs. TrustedBSD is not only used by FreeBSD, but also by Apple to implement seatbelt in Mac OS X and iOS [33].

FlaskDroid [9] also shares motivation with ASM. It provides an SELinux-style Type Enforcement (TE) policy language for extending Android security. FlaskDroid also allows third-party application developers to specify TE policies to protect their applications. However, FlaskDroid is limited to TE access control policies. By providing a programmable interface, ASM enables an extensible interface that allows not only TE, but also novel security models not yet invented. Specifically, we believe the ability to replace data values will become vital in protecting new operating systems such as Android.

Concurrent to and independent of our work on ASM, the Android Security Framework (ASF) [4] also provides an extensible interface to implement security modules. ASM and ASF are conceptually very similar: both promote the need for a programmable interface, authorization hooks that replace data (called edit automata in ASF), and third-party hooks (via *callModule()* in ASF). However, their individual approaches differ. A key difference is that ASM seeks to ensure existing security guarantees (Goal **G2**), whereas ASF assumes the module writer is completely trusted (e.g., can load kernel modules). Goal **G2** is motivated by our vision of enterprise IT and researchers loading ASM apps on production phones without root access. ASF does not support this vision. Furthermore, a vulnerable ASF module can undermine secrecy and integrity of the system and all installed applications. That said, ASF does provide expressibility that ASM does not. Specifically, ASF provides a programmable interface to adding inline reference monitors to apps. While IRMs run the risk of breaking apps, they do support sub-application policies that ASM cannot express (e.g., forcing an app to use *https* over *http*).

Finally, Section 5.4 identified the AppOps security enhancement that is currently under development in AOSP. AppOps adds authorization hooks throughout the Android OS. However, AppOps does not provide a programmable interface for enhancing OS security. Instead,

we envision separating the authorization hooks from AppOps and implementing AppOps as an ASM app. A similar process occurred during the creation of LSM when it was split away from SELinux.

8 Conclusion

This paper has presented the Android Security Modules framework as a programmable interface for extending Android's security. While similar reference monitor interfaces have been proposed for Linux and TrustedBSD, ASM is novel in how it addresses the semantically rich OS APIs provided by new operating systems such as Android. We studied over a dozen research proposals that enhance Android security to motivate the reference monitor interface hooks provided by ASM. Of particular note is the ability for hooks to replace data, as well as for third-party application developers to define new hooks.

ASM promotes the creation of novel security enhancements to Android without restricting OS consumers (e.g., consumers, enterprise, government) to specific policy languages (e.g., type enforcement). ASM currently allows researchers with the ability to recompile Android to rapidly prototype novel reference monitors without needing to consider authorization hook placement. If ASM is adopted into the AOSP source code, it potentially allows researchers and enterprise IT to add new reference monitors to production Android devices without requiring root access, a significant limitation of existing bring-your-own-device solutions.

Acknowledgements

This material is based upon work supported by the National Science Foundation. Adwait Nadkarni and William Enck were partially supported by NSF grants CNS-1253346 and CNS-1222680. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] android-apktool. `https://code.google.com/p/android-apktool/`.

[2] ANDERSON, J. P. Computer Security Technology Planning Study. ESDTR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I, DITCAD-758206. Vol. II DITCAD-772806).

[3] AppArmor. `http://apparmor.net`. Accessed January 2014.

[4] BACKES, M., BUGIEL, S., GERLING, S., AND VON STYP-REKOWSKY, P. Android Security Framework: Enabling Generic and Extensible Access Control on Android. Tech. Rep. A/01/2014, Saarland University, April 2014.

[5] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND STYP-REKOWSKY, P. AppGuard - Enforcing User Requirements on Android Apps. In *Proceedings of TACAS*. Springer Berlin Heidelberg, 2013.

[6] BERESFORD, A. R., RICE, A., SKEHIN, N., AND SOHAN, R. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of HotMobile* (2011).

[7] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Tech. Rep. TR-2011-04, Technische Universitat Darmstadt, Center for Advanced Security Research Darmstadt, Darmstadt, Germany, Apr. 2011.

[8] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)* (2011).

[9] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the USENIX Security Symposium* (2013).

[10] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of ISC* (2010).

[11] DAVIS, B., AND CHEN, H. RetroSkeleton: Retrofitting Android Apps. In *Proceedings of MobiSys* (2013).

[12] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium* (2011).

[13] DOMOBILE LAB. AppLock. `https://play.google.com/store/apps/details?id=com.domobile.applock`.

[14] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of USENIX OSDI* (2010).

[15] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of ACM CCS* (2009).

[16] FELT, A. P., EGELMAN, S., FINIFTER, M., AKHAWE, D., AND WAGNER, D. How to Ask For Permission. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)* (2012).

[17] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: user attention, comprehension, and behavior. In *Proceedings of SOUPS* (2012).

[18] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (Aug. 2011).

[19] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of ACM CCS* (2011).

[20] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)* (2012), ACM.

[21] MCDANIEL, P., AND PRAKASH, A. Methods and limitations of security policy reconciliation. In *Proceedings of IEEE S&P* (2002), IEEE Computer Society.

[22] NADKARNI, A., AND ENCK, W. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of ACM CCS* (2013).

[23] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of ASIACCS* (2010).

[24] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the Annual Computer Security Applications Conference* (2009).

[25] QUALCOMM. Trepn Profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

[26] RAO, V., AND JAEGER, T. Dynamic Mandatory Access Control for Multiple Stakeholders. In *Proceedings of ACM SACMAT* (2009), ACM.

[27] SCHAUFLER, C. [PATCH v13 0/9] LSM: Multiple concurrent LSMs. https://lkml.org/lkml/2013/4/23/307, 2013.

[28] Security-Enhanced Linux. http://selinuxproject.org. Accessed January 2014.

[29] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of NDSS* (2013).

[30] The Smack Project. http://schaufler-ca.com/. Accessed February 2014.

[31] TOMOYO Linux. http://tomoyo.sourceforge.jp. Accessed February 2014.

[32] WATSON, R. N. M. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track* (2001).

[33] WATSON, R. N. M. A Decade of OS Access-Control Extensibility. *Communications of the ACM 56*, 2 (Feb. 2013).

[34] WRITE, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the USENIX Security Symposium* (2002).

[35] XU, R., SAIDI, H., AND ANDERSON, R. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the USENIX Security Symposium* (2012).

[36] Yama lsm. https://www.kernel.org/doc/Documentation/security/Yama.txt. Accessed February 2014.

[37] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. W. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing* (June 2011).

Brahmastra: Driving Apps to Test the Security of Third-Party Components

Ravi Bhoraskar[1,2], *Seungyeop Han*[2], *Jinseong Jeon*[3], *Tanzirul Azim*[4], *Shuo Chen*[1], *Jaeyeon Jung*[1], *Suman Nath*[1], *Rui Wang*[1], *David Wetherall*[2]

[1] *Microsoft Research*
[2] *University of Washington*
[3] *University of Maryland, College Park*
[4] *University of California, Riverside*

Abstract

We present an app automation tool called Brahmastra for helping app stores and security researchers to test third-party components in mobile apps at runtime. The main challenge is that call sites that invoke third-party code may be deeply embedded in the app, beyond the reach of traditional GUI testing tools. Our approach uses static analysis to construct a page transition graph and discover execution paths to invoke third-party code. We then perform binary rewriting to "jump start" the third-party code by following the execution path, efficiently pruning out undesired executions. Compared with the state-of-the-art GUI testing tools, Brahmastra is able to successfully analyse third-party code in 2.7× more apps and decrease test duration by a factor of 7. We use Brahmastra to uncover interesting results for two use cases: 175 out of 220 children's apps we tested display ads that point to web pages that attempt to collect personal information, which is a potential violation of the Children's Online Privacy Protection Act (COPPA); and 13 of the 200 apps with the Facebook SDK that we tested are vulnerable to a known access token attack.

1 Introduction

Third-party libraries provide a convenient way for mobile application developers to integrate external services in the application code base. Advertising that is widely featured in "free" applications is one example: 95% of 114,000 popular Android applications contain at least one known advertisement library according to a recent study [22]. Social media add-ons that streamline or enrich the user experience are another popular family of third-party components. For example, `Facebook Login` lets applications authenticate users with their existing Facebook credentials, and post content to their feed.

Despite this benefit, the use of third-party components is not without risk: if there are bugs in the library or the way it is used then the host application as a whole becomes vulnerable. This vulnerability occurs because the library and application run with the same privileges and without isolation under existing mobile application models. This behavior is especially problematic because a number of third-party libraries are widely used by many applications; any vulnerability in these libraries can impact a large number of applications. Indeed, our interest in this topic grew after learning that popular SDKs provided by Facebook and Microsoft for authentication were prone to misuse by applications [30], and that applications often make improper use of Android cryptography libraries [20].

In this paper, we present our solution to the problem of *third-party component integration testing at scale*, in which one party wishes to test a large number of applications using the same third-party component for a potential vulnerability. To be useful in the context of mobile app stores, we require that a successful solution test many applications without human involvement. Observe that it is *not* sufficient to simply test the third-party library for bugs in isolation. This is because vulnerabilities often manifest themselves due to the interaction of the application and the third-party component. Thus our focus is to develop tools that enable testers to observe *in situ* interactions between the third-party component and remote services in the context of a specific application at runtime.

We began our research by exploring automated runtime analysis tools that drive mobile UIs (e.g., [5, 23, 26]) to exercise the third-party component, but quickly found this approach to be insufficient. Although these tools are effective at executing *many* different code paths, they are often unable to reach *specific* interactions deep within the applications for a number of reasons that we explore within this paper. Instead, our approach leverages the structure of the app to improve test hit rate and execution speed. To do this, we characterize an app by statically building a graph of its pages and transitions between them. We then use path information from the graph to guide the runtime execution towards the third-

party component under test. Rather than relying on GUI manipulation (which requires page layout analysis) we rewrite the application under test to directly invoke the callback functions that trigger the desired page transitions.

We built Brahmastra to implement our approach for Android apps. Our tool statically determines short execution paths, and dynamically tests them to find one that correctly invokes a target method in the third-party library. At this stage, behavior that is specific to the library is checked. Because our techniques do not require human involvement, Brahmastra scales to analyze a large number of applications. To show the benefits of our approach, we use our tool for two new studies that contribute results to the literature: 1) checking whether children's apps that source advertisements from a third-party comply with COPPA privacy regulations; and 2) checking that apps which integrate the Facebook SDK do not have a known security vulnerability [30].

From our analysis of advertisements displayed in 220 kids apps that use two popular ad providers, we find that 36% apps have displayed ads whose content is deemed inappropriate for kids—such as offering free prizes, or displaying sexual imagery. We also discover that 80% apps have displayed ads with landing pages that attempt to collect personal information from the users, such as name, address, and online contact information—which can be a violation of the Children's Online Privacy Protection Act [6]. Apart from creating an unsafe environment for kids, this also leaves the app developers vulnerable to prosecution, since they are considered liable for all content displayed by their app.

For our analysis of a vulnerability in third party login libraries, we run a test case proposed by Wang et al. [30] against 200 Android apps that bundle Facebook SDK. We find that 13 of the examined apps are vulnerable.

Contributions: We make two main contributions. The first is Brahmastra, which embodies our hybrid approach of static and dynamic analysis to solve the third-party component integration testing problem for Android apps. We discuss our approach and key techniques in §4 and their implementation in §5. We show in §6 that our techniques work for a large fraction of apps while existing tools such as randomized testing (Monkey) often fail. We have made the static analysis part of Brahmastra available at `https://github.com/plum-umd/redexer`.

Our second contribution is an empirical study of two security and privacy issues for popular third-party components. We find potential violations of child-safety laws by ads displayed in kids apps as discussed in §7; several apps used in the wild display content in potential violation of COPPA due to the behavior of embedded components. We find that several popular Android apps are vulnerable to the Facebook access token attack as discussed in §8; A Facebook security team responded immediately to our findings on 2/27/2014 and had contacted the affected developers with the instructions to fix.

2 Background

As our system is developed in the context of Android, we begin by describing the structure of Android apps and support for runtime testing.

Android app structure: An Android app is organized as a set of pages (e.g., Figure 1) that users can interact with and navigate between. In Android, each page is represented by an `activity` object. Each activity class represents one kind of page and may be initialized with different data, resulting in different `activity instances`. We use the terms page and activity instance interchangeably. Each page contains various GUI elements (e.g., buttons, lists, and images), known as `views`. A view can be associated with a callback function that is invoked when a user interacts with the view. The callback function can instantiate a new activity by using a late binding mechanism called `intent`. An intent encapsulates the description of a desired action (e.g., start a target activity) and associated parameters. The main `activity` (or the first page) of an app, defined in its manifest file, is started by the application launcher by passing a `START` intent to it.

For example, in Figure 1, clicking the "Done" button on activity `A1` invokes its event handler, which calls a callback function defined by the app developer. The callback constructs an intent to start activity `A2` with necessary parameter `P12`. The `Activity Manager` then constructs an instance of `A2`, and starts it with `P12` as parameters. We refer to the documentation of Android internals for more details [2].

Automated dynamic analysis: Recent works have used a class of automation tools, commonly called a Monkey, that, given a mobile app binary, can automatically execute it and navigate to various parts (i.e., states) of the app. Examples include PUMA [23], DECAF [25], AppsPlayground [26], A3E [14], and VanarSena [27]. A Monkey launches the app in a phone or an emulator, interacts with it by emulating user interactions (e.g., clicking a button or swiping a page) to recursively visit various pages, and performs specific tasks (e.g., checking ad frauds in the page or injecting faults) on each page.

In Figure 1, a Monkey may be able to visit the sequence of states `A1` $\rightarrow$ `A2` $\rightarrow$ `A3`$\rightarrow$ `A4` if it knows the right UI actions (e.g., type in mother's name and select "Due Date" in `A1`) to trigger each transition. However, if Monkey clicks a button in `A3` other than "Account", the app would navigate to a different activity. If the goal of testing is to invoke specific methods (e.g., Facebook login as shown in the example), then without knowing the structure of the app, a Monkey is likely to wander around

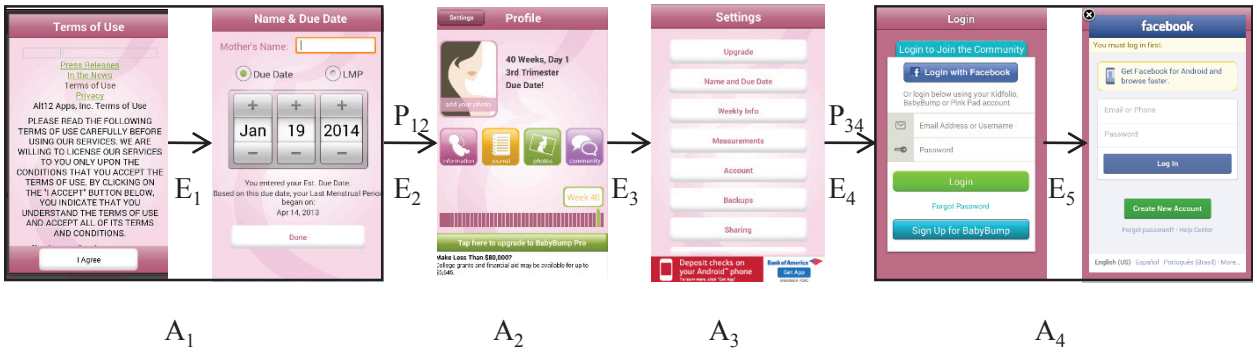


Figure 1: Activity sequences of `com.alt12.babybumpfree` that invoke Facebook single sign-on window in the forth activity (`A4`): Clicking "I Agree" (`E1`) then clicking "Done" (`E2`) opens up `A2` with the parameter, `fromLoader : tru` (`P12`). Clicking "Settings" (`E3`) in `A2` opens up the settings activity, `A3` and then clicking "Account" (`E4`) opens up the login activity, `A4` with the parameter, `WHICHCLASS : com.alt12.babybumpcore.activity.settings.Settings`. Finally, clicking "Login with Facebook" (`E5`) opens up the sign-on window within the same activity, `A4`.

many activities until it reaches `A4`, if it ever does.

3 Problem and Insights

Our goal is to develop the ability to automatically and systematically test a large set of mobile apps that embed a specific third-party component for a potential vulnerability associated with the use of that component. This ability would let app store operators rapidly vet apps to contain security vulnerabilities caused by popular libraries. It would let component developers check how apps use or misuse their interfaces. It would also let security researchers such as ourselves empirically assess vulnerabilities related to third-party libraries.

A straightforward approach is to use existing Monkeys. Unfortunately, this approach does not work well: it often fails to exercise the target third-party component of the app under test. Although recent works propose techniques to improve various types of *coverages*, computed as the fraction of app activities or methods invoked by the Monkey, coverage still remains far from perfect [26, 14, 13, 25, 27]. Moreover, in contrast to traditional coverage metrics, our success metric is binary for a given app indicating whether the target third-party component (or a target method in it) is invoked (i.e., *hit*) or not (i.e., *miss*). Our experiments show that even a Monkey with a good coverage can have a poor hit rate for a target third-party component that may be embedded deep inside the app. We used an existing Monkey, PUMA that reports a $> 90\%$ activity coverage compared to humans [23], but in our experiments it was able to invoke a target third-party component only in 13% of the apps we tested (see §6 for more details). On a close examination, we discovered several reasons for this poor hit rate of existing Monkeys:

R1. Timeout: A Monkey can exhaust its time budget before reaching the target pages due to its trial-and-error search of the application, especially for apps with many pages that "blow up" quickly.

R2. Human inputs: A Monkey is unable to visit pages that are reached after entering human inputs such as login/password, or gestures beyond simple clicks that the automated tester cannot produce.

R3. Unidentified elements: A Monkey fails to explore clickable UI elements that are not visible in the current screen (e.g., hidden in an unselected tab) or are not activated yet (e.g., a "Like" button that is activated only after the user registers to the app) or are not identified by underlying UI automation framework (e.g., nonstandard custom control).

R4. Crashes: By stressing the UI, a Monkey exacerbates app crashes (due to bugs and external dependencies such as the network) that limit exploration.

Note that, unlike existing Monkeys, our goal is not to exhaustively execute all the possible code paths but to execute *particular code paths* to invoke methods of interest in the third-party library. Therefore, our insight is to improve coverage by leveraging ways how third party components are integrated with application code base. These components are incorporated into an app at the activity level. Even if the same activity is instantiated multiple times with different contents, third-party components typically behave in the same way in all those instantiations. This allows us to restrict our analysis at the level of *activity* rather than *activity instances*. Further, even if an app contains a large number of activities, only a small number of them may actually contain the third-party component of interest. Invoking that compo-

nent requires successfully executing any and only one of those activities.

Using this insight, our testing system, Brahmastra, uses three techniques described below to significantly boost test hit rate and speed compared to a Monkey that tries to reach all pages of the app.

Static path pruning: Brahmastra considers only the "useful" paths that eventually invoke the target third-party methods and ignores all other "useless" paths. In Figure 1, Brahmastra considers the execution path `A1` → `A2` → `A3`→ `A4` for exploration and ignores many other paths that do not lead to a target activity, `A4`.

Such useful paths need to be identified statically *before* dynamic analysis is performed. The key challenges in identifying such paths by static analysis arise due to highly asynchronous nature of Android apps. We discuss the challenges and our solution in §4.

Dynamic node pruning: Brahmastra opportunistically tries to start from an activity in the middle of the path. If such "jump start" is successful, Brahmastra can ignore all preceding activities of the path. For example, in Figure 1, Brahmastra can directly start activity `A3`, which can lead to the target activity `A4`.

Dynamic node pruning poses several challenges — first, we need to enable jump-starting an arbitrary activity directly. Second, jump starting to the target activity may fail due to incorrect parameters in the intent, in which case we need to find a different activity that is close to the target, for which jump start succeeds. We discuss these in detail in next section.

Self-execution of app: Brahmastra rewrites the app binary to automatically call methods that cause activity transitions. The appropriate methods are found by static analysis. In Figure 1, instead of clicking on the button with label "Done" in `A1`, Brahmastra would invoke the `onClick()` method that would make the transition from `A1` to `A2`. The advantage over GUI-driven automation is that it can discover activity-transitioning callbacks even if they are invisible in the current screen.

In summary, our optimizations can make dynamic analysis fast by visiting only a small number of activities of an app. More importantly, they also improve the test hit rate of such analysis. Faster analysis helps to avoid any timeouts (**R1**). Dynamic node pruning can bypass activities that require human inputs (**R2**). In Figure 1, Brahmastra can jump to `A3` and bypass `A1` that requires selecting a *future* due date. Intent-driven navigation helps Brahmastra to make transitions where a Monkey fails due to unidentified GUI elements (**R3**). Finally, visiting fewer activities reduces the likelihood of crashes (**R4**). We quantitatively support these claims in §6.

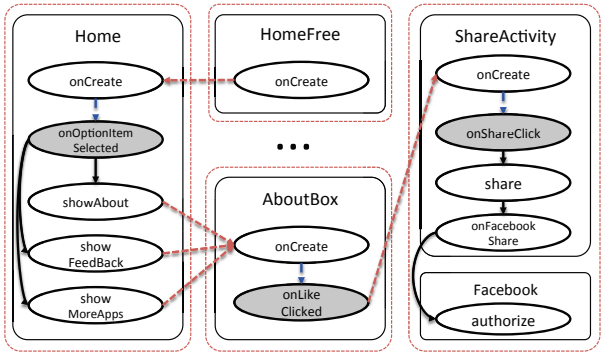


Figure 2: A simplified call graph of `ch.smalltech.battery.free` that shows multiple transition paths composed of multiple activities. Boxes and ovals represent classes and methods. Solid edges correspond to synchronous calls; (red) dotted edges indicate activity transitions; and (blue) dashed edges represent implicit calls due to user interactions. Three different paths starting from `Home.onOptionItemSelected()` reach `AboutBox.onCreate()` and share the remaining part.

4 Design

Brahmastra requires as input: a test application binary; the names of target methods to be invoked within the context of the application; and the plug-in of a specific security analysis to run once the target method is reached. Our system is composed of three parts:

1. *Execution Planner* statically analyzes the test app binary and discovers an execution path to invoke the target third-party method.
2. *Execution Engine* receives execution paths from the Planner and launches the test app in one or multiple emulators and automatically navigates through various pages according to the execution path.
3. *Runtime Analyzer* is triggered when the test app invokes the target method. It captures the test app's runtime state (e.g., page content, sensors accessed, network trace) and runs the analysis plug-in.

4.1 Execution Planner

The job of the Execution Planner is to determine: (1) the activities that invoke the target third-party method; and (2) the method-level execution paths that lead to the target activities. To accomplish these tasks, we statically analyze the app binary to construct a *call graph* that encompasses its activities and interactions that cause activity transitions.

Constructing call graph: A call graph is a graph where vertices are methods and edges are causal relationship between method invocation. More precisely, there exists

```
ImageButton b = (ImageButton)
              findViewById(R.id.b1);
b.setOnClickListener(new OnClickListener() {
  public void onClick(View v) {
    ...
  }});
```

Figure 3: Example of a programmatic handler registration. `onClick()` is bound to `setOnClickListener()`

an edge from method m_1 to m_2 if m_1 invokes m_2. Based on how m_2 is invoked by m_1, there are three types of edges: (1) synchronous edges, if m_1 directly calls m_2, (2) asynchronous edges, if m_1 invokes m_2 asynchronously, and (3) activity transition edges, if m_1 starts an activity that automatically calls m_2. Figure 2 depicts a call graph of one real app.

While synchronous edges can be identified easily by scanning the app binary code, discovering other edges can be difficult. To find activity transition edges, we rely on the fact that one activity can start another activity by generating an intent and passing it to the `startActivity()` method. We perform constant propagation analysis [12] so as to track such intent creations and detect activity transitions. We also conduct class hierarchy analysis [19] to conservatively determine possible receiver types for dynamic dispatch, where the target call sites depend on the runtime types of the receivers.

To discover asynchronous edges, we need to consider all the different ways asynchronous methods can be invoked by a mobile app:

1. Programmatic handler registrations: These are callbacks explicitly bound to methods (e.g., event handler of GUI elements) within the code. Figure 3 shows an example.
2. XML-based handler registrations: These are callbacks specified in the layout or resource XML files. Figure 4 shows an example.
3. Lifetime methods: These are methods provided by the underlying framework that automatically make transitions to other methods on specific events. Examples are splash screens and message boxes that transition to next activities after a timeout or after user acknowledgment, respectively.

To discover the first and third types, we use constant propagation analysis to trace callbacks attached to various event handlers. To handle the second case, we parse layout XML files corresponding to each activity to figure out the binding between UI elements and callback methods.

Efficient call graph computation: A call graph can be extremely large, thus computing the entire call graph can

```
// layout/about_box_share.xml
<Button android:id="@id/mShareFacebook"
  style="@style/ABB_Black_ShareButton" ... />
<Button android:id="@id/mShareTwitter"
  style="@style/ABB_Black_ShareButton" ... />
// values/styles.xml
<style name="ABB_Black_ShareButton ... >
  <item name="android:onClick">onShareClick</item>
</style>
// ch.smalltech.common.feedback.ShareActivity
public void onShareClick(View v){
  // different behavior depending on argument v
}
```

Figure 4: Example of a XML-based handler registration observed from `ch.smalltech.battery.free`. Two buttons share the `onShareClick` callback. The binding between `onShareClick` and `setOnClickListener` of each button can be determined through layout and styles XML files.

be very expensive. For example, the app shown in Figure 1 declares 74 activities in the manifest; we find at least 281 callbacks over 452 registering points; and its call graph is composed of 1,732 nodes and 17,723 edges. To address this, we use two optimizations to compute a *partial* call graph that includes target methods and the start activity methods. First, we exclude system's static libraries and other third-party libraries that are not related to the target methods. Second, we search transition paths backwards on call graph. We pinpoint call sites of target methods while walking through bytecodes. We then construct a partial call graph, taking into accounts component transitions via intent and bindings between views and listeners. Finally, starting from the call sites, we build backward transition paths, until public components including the main activity are reached. If failed, partial paths collected at the last phase will be returned.

Determining target activity(s): Given the call graph and a target method, we determine the activities that invoke the method as follows. From the call graph, we can identify the activity boundaries such that all methods within the same boundary are invoked by the same activity. Since an activity can be started only through an activity transition edge in the call graph, any maximal connected component whose edges are either synchronous or asynchronous define the boundary of an activity. In Figure 2, bigger rectangles denote the activity boundaries. Given the boundaries, we identify the activities that contain the target method.

Finding activity transition paths: Reaching a target activity from the start activity may require several transitions between multiple activities. For example, in Figure 2, navigating from the start activity (`HomeFree`) to a target activity (`ShareActivity`) requires three transi-

tions. This implies that Brahmastra requires techniques for automatic activity transitions, which we describe in the next subsection. Second, a target activity may be reachable via multiple transition paths. While the shortest path is more attractive for fast exploration, the path may contain blocking activities and hence not executable by Brahmastra. Therefore, Brahmastra considers all transition paths (in increasing order of their length); if execution of a short path fails, it tries a longer one.

Given the call graph G, the Planner computes a small set P of acyclic transition paths that the Execution Engine need to consider. P includes a path if and only if it terminates at a target activity without a cycle and is not a suffix of any other path in P. This ensures that P is useful, complete (i.e., Execution Engine does not need to consider any path not in P), and compact. For instance Figure 5 shows one out of three paths contained in P.

```
     HomeFree;.onCreate
---> Home;.onCreate
-#-> Home;.onOptionsItemSelected
---> Home;.showAbout
---> AboutBox;.onCreate
-#-> AboutBox;.onLikeClicked
---> ShareActivity;.onCreate
-#-> ShareActivity;.onShareClick
---> ShareActivity;.share
---> ShareActivity;.onFacebookShare
```

Figure 5: An example path information for `ch.smalltech.battery.free` Dashed arrows stand for explicit calls or activity transition, whereas arrows with a hash tag represent implicit invocations, which are either callbacks due to user interactions or framework-driven callbacks, such as lifecycle methods.

P can be computed by breadth-first traversals in G, starting from each target activity and traversing along the reverse direction of the edges.

4.2 Execution Engine

The useful paths P produced by the Execution Planner already give an opportunity to prune exploration: Brahmastra considers only paths in P (and ignore others), and for each path, it can simply navigate through its activities from the beginning of the path (by using techniques described later). Exploration can stop as soon as a target method is invoked.

Rewriting apps for self-execution: One might use a Monkey to make activity transitions along useful paths. Since a Monkey makes such transitions by interacting with GUI elements, this requires identifying mapping between GUI elements and transitioning activities and interact with only the GUI elements that make desired transitions.

We address this limitation with a technique we develop called *self execution*. At a high level, we rewrite app binaries to insert code that automatically invokes the callbacks that trigger desired activity transitions, even if their corresponding GUI elements are not visible. Such code is inserted into all the activities in a useful path such that the rewritten app, after being launched in a phone or an emulator, would automatically make a series of activity transitions to the target activity, without any external interaction with its GUI elements.

Jump start: Brahmastra goes beyond the above optimization with a node pruning technique called "jump start". Consider a path $p = (a_0, a_1, \ldots, a_t)$, where a_t is a target activity. Since we are interested only in the target activity, success of Brahmastra is not affected by what activity a_i in p the execution starts from, as long as the last activity a_t is successfully executed. In other words, one can execute any suffix of p without affecting the hit rate. The jump start technique tries to execute a suffix — instead of the whole — useful path. This can improve Brahmastra's speed since it can skip navigating through few activities (in the prefix) of a useful path. Interestingly, this can also improve the hit rate of Brahmastra. For example, if the first activity a_0 requires human inputs such as user credentials that an automation system cannot provide, any effort to go beyond state a_0 will fail.

Note that directly executing an activity $a_i, i > 0$, without navigating to it from the start activity a_0, may fail. This is because some activities are required to be invoked with specific intent parameters. In such cases, Brahmastra tries to jump start to the previous activity a_{i-1} in the path. In other words, Brahmastra progressively tries to execute suffixes of useful paths, in increasing order of lengths, until the jump start succeeds and the target activity a_t is successfully reached or all the suffixes are tried.

Algorithm 1 shows the pseudocode of how execution with jump start works. Given the set of paths, the algorithm first generates suffixes of all the paths. Then it tries to execute the suffixes in increasing order of their length. The algorithm returns true on successful execution of any suffix. Note that Algorithm 1 needs to know if a path suffix has been successfully executed (line 9). We inject lightweight logging into the app binary to determine when and whether target methods are invoked at runtime.

4.3 Runtime Analyzer

Runtime Analyzer collects various runtime states of the test app and makes it available to custom analysis plug-ins for scenario-specific analysis. Runtime states include UI structure and content (in form of a DOM tree) of the current app page, list of system calls and sensors invoked by the current page, and network trace due to the current page. We describe two plug-ins and analysis results in

Algorithm 1 Directed Execution

```
1:  INPUT: Set of useful paths P from the Planner
2:  OUTPUT: Return true if execution is successful
3:  S ← set of suffixes of all paths in P
4:  for i from 0 to ∞ do
5:      S_i ← set of paths of length i in S
6:      if S_i is empty then
7:          return false
8:      for each path suffix p in S_i do
9:          if Execute(p) = true then
10:             return true
11:
12: return false
```

```
1  // { v3 →this }
2  new-instance v0, Landroid/content/Intent;
3  // { v0 →Intent(), ... }
4  const-class v1, ...AboutBox;
5  // { v1 →Clazz(AboutBox), ... }
6  invoke-direct {v0, v3, v1}, ...Intent;.<init>
7  // { v0 →Intent(AboutBox), ... }
8  invoke-virtual {v3, v0}, ...;.startActivity
9  // { ... }
```

Figure 6: An example bytecode of activity transition excerpted from `ch.smalltech.battery.free`. Mappings between bytecode represent data-flow information, which shows what values registers *must* have. Only modified or newly added values are shown.

later sections.

5 Implementation of Brahmastra

We implement Brahmastra for analyzing Android apps, and use the tool to perform two security analyses which we will describe in §7 and §8. This section describes several highlights of the tool, along with practical challenges that we faced in the implementation process and how we resolved them.

5.1 Execution Planner

Static analyses for constructing a call graph and finding transition paths to target methods are performed using Redexer [24], a general purpose bytecode rewriting framework for Android apps. Redexer takes as input an Android app binary (APK file) and constructs an in-memory data structure representing `DEX` file for various analyses. Redexer offers several utility functions to manipulate such `DEX` file and provides a generic engine to perform data-flow analysis, along with call graph and control-flow graph.

For special APIs that trigger activity transitions, e.g., `Context.startActivity()`, we perform constant propagation analysis (see Appendix A for details) and identify a target activity stored inside the intent. Figure 6 depicts example bytecode snippets that create and initialize an intent (lines 2 and 6), along with the target activity (line 4), and starts that activity via `startActivity()` (line 8). Mappings between each bytecode show how we accumulate data-flow information, from empty intent through class name to intent with the specific target activity. We apply the same analysis to bindings between views and listeners.

5.2 App Rewriting

We use the Soot framework [29] to perform the bytecode rewriting that enables self execution. Dexpler [7] converts an Android app binary into Soot's intermediate representation, called Jimple, which is designed to ease analysis and manipulation. The re-writing tool is composed of Soot's class visitor methods and an Android XML parser. Given an app binary and an execution path, the rewriter generates a rewritten binary which artificially invokes a callback method upon the completion of the exercising the current activity, triggering the next activity to be launched. The inserted code depends on the type of the edge found by the Planner (Recall three kinds of asynchronous edges described in §4.1). For programmatic and XML-based registrations, the rewriter finds the `view` attached to it — by parsing the activity code, and the manifest respectively — and invokes the appropriate UI interaction on it after it has completed loading. Lifetime methods are invoked by the Android framework directly, and the rewriter skips code insertion for these cases. In other cases, the rewriter inserts a timed call to the transition method directly, to allow the activity and any dependencies of the method to load completely.

5.3 Jump Start

Jump start requires starting an activity even if it is not defined as the `Launcher` activity in the app. To achieve that, we manipulate the manifest file of the Android app. The `Intent.ACTION_MAIN` entry in the manifest file declares activities that Android activity manager can start directly. To enable jump start, we insert an `ACTION_MAIN` entry for each activity along the path specified, so that it can be started by the Execution Engine. Manifest file also declares an intent filter, which determines the sources from which an activity may be started, which we modify to allow the Execution Engine to launch the activity. The Engine then invokes desired activity by passing an intent to it. We use the Android Debug Bridge (ADB) [4] for performing jump start. ADB allows us to create an intent with the desired parameters and target, and then passes it to the Android Activity Manager. The activity manager in turn loads the appropriate app data and invokes the specified activity. Starting the (jump started) activ-

ity immediately activates self execution from that activity onwards.

6 Evaluation of Brahmastra

We evaluate Brahmastra in terms of two key metrics: (1) hit rate, i.e., the fraction of apps for which Brahmastra can invoke any of target methods, and (2) speed, i.e., time (or number of activity transitions) Brahmastra takes to invoke a target method in an app for which Brahmastra has a hit. Since we are not aware of any existing tool that can achieve the same goal, we compare Brahmastra against a general Android app exploration tool called, PUMA [23]. This prototype is the best-performing Monkey we were able to find that is amenable to experimentation. In terms of speed and coverage, PUMA is far better than a basic "random" Monkey. PUMA incorporates many key optimizations in existing Monkeys such as AppsPlayground [26], A3E [14], and VanarSena [27] and we expect it to perform at least on a par with them.

6.1 Experiment Methodology

Target method: For the experiments in this section, we configure Brahmastra to invoke authentication methods in the Facebook SDK for Android.[1] We choose Facebook SDK because this is a popular SDK and its methods are often invoked only deep inside the apps. Using the public documentation for the Facebook SDK for Android, we determined that it has two target methods for testing. Note that apps in our dataset use the Facebook SDK version 3.0.2b or earlier[2]

```
Lcom/facebook/android/Facebook;->authorize
Lcom/facebook/android/Facebook;->dialog
```

Figure 7: Target methods for evaluation

Apps: We crawled 12,571 unique apps from the Google Play store from late December 2012 till early January 2013. These apps were listed as 500 most popular free apps in each category provided by the store at the time. Among them, we find that 1,784 apps include the Facebook SDK for Android. We consider only apps that invoke the authentication method—Over 50 apps appear to have no call sites to Facebook APIs, and over 400 apps use the API but do not invoke any calls related to authorization. We also discard apps that do not work with our tool chain, e.g., crash on the emulator or have issues with `apktool` [1] since our analysis depends on the disassembled code of an apk file. This leaves us with 1,010 apps.

[1] `https://developers.facebook.com/docs/android/login-with-facebook`

[2] The later version of Facebook SDK was released in the middle of data collection and appears to use different methods to display a login screen. However, we find that almost no apps in our data set had adapted the new version yet.

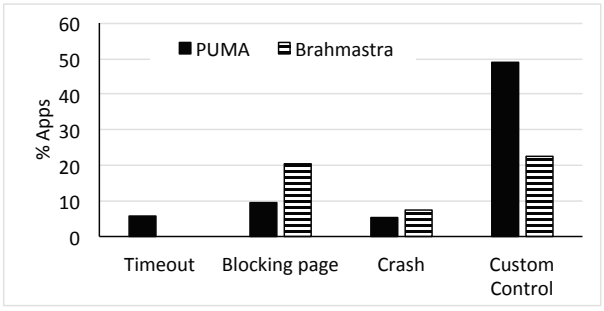


Figure 8: Failure causes of Brahmastra and PUMA.

App execution: In order to determine if Brahmastra or PUMA is able to reach a program point that invokes the target method, we instrument apps. The instrumentation detects when any of the target methods are invoked during runtime, by comparing signatures of executing methods with signatures of target methods. For Brahmastra, we consider only 5 of all paths generated by the Execution Planner. For PUMA, we explore each app for up to 250 steps; higher timeouts significantly increase overall testing time for little gain in hit rate.

6.2 Hit Rate

In our experiments, PUMA was able to successfully invoke a target method in 127 apps (13%). Note that PUMA's hit rate is significantly lower than its activity coverage ($>$ 90% compared to humans) reported in [23], highlighting the difficulty in invoking specific program points deep inside an app. In contrast, Brahmastra was successfully able to invoke a target method in 344 (34%) apps, a 2.7$\times$ improvement over PUMA. A closer examination of our results, as shown in Table 1, reveals that Brahmastra's technique can help circumventing all the root causes for PUMA's poor hit rate as mentioned in §3.

We now investigate why PUMA and Brahmastra sometimes fail to invoke the target method. For PUMA, this is due to the aforementioned four cases. Figure 8 shows the distribution of apps for which PUMA fails due to specific causes. As shown, all the causes happen in practice. The most dominant cause is the failure to find UI controls to interact with, which is mostly due to complex UI layouts of the popular apps we tested. Figure 8 also shows the root causes for Brahmastra's failure. The key reasons are as follows:

Blocking page: Even if jump start succeeds, successive activity transition may fail on a blocking page. Brahmastra fails for 20% of the apps due to this cause. We would like to emphasize that Brahmastra experiences more blocking pages than PUMA only because Brahmastra explores many paths that PUMA does not (e.g., because those paths are behind a custom control that

Case	Apps
R1: Timeout in PUMA, success in Brahmastra	62%
R2: Blocking page in PUMA, success in Brahmastra	48%
R3: Unknown control in PUMA, success in Brahmastra	43%
R4: Crash in PUMA, success in Brahmastra	30%

Table 1: % of apps for which Brahmastra succeeds but PUMA fails due to various reasons mentioned in §3.

PUMA cannot interact with, but Brahmastra can find and invoke the associated callback) and many of these paths contain blocking pages. If PUMA tried to explore those paths, it would have failed as well due to these blocking pages.

Crash: Jump start can crash if the starting activity expects specific parameters in the intent and Brahmastra fails to provide that. Brahmastra fails for 7% of the apps due to this cause.

Custom components: Execution Planner may fail to find useful paths if the app uses custom components[3], which can be used to override standard event handlers, thus breaking our model of standard Android apps. Without useful paths, Brahmastra can fail to invoke the target methods. In our experiments, this happens with 16% of the apps. We leave as future work a task to extend Execution Planner to handle custom components. We find that PUMA also failed 91% on these apps, proving the difficulty of navigating apps with custom components. In fact, PUMA suffers much more than Brahmastra due to custom components.

Improving the hit rate: There are several ways we can further improve the hit rate of Brahmastra. First, 16% failures of Brahmastra come because the static analysis fails to identify useful paths. A better static analysis that can discover more useful paths can improve Brahmastra's hit rate. Second, in our experiments, Brahmastra tried only up to 5 randomly selected useful paths to invoke the target method and gave up if they all failed. In many apps, our static analysis found many tens of useful paths, and our results indicate that the more paths we tried, the better was the hit rate. More specifically, Brahmastra succeeded for 207 apps after considering only one path, and for 344 apps after considering up to five paths. This suggests that considering more paths is likely to improve the hit rate. Additionally, we should select the paths to avoid any nodes or edges for which exploration failed in previously considered paths instead of choosing them randomly. In 72 apps, Brahmastra was unable to find the binding between a callback method and the UI element associated with it, causing it to fall back on a direct invocation of the callback method. A better static analysis can help in this case as well. In 22 apps, Brahmastra deemed a page blocked due to UI elements in the Android SDK (e.g., list elements) whose behavior was not encoded in the instrumentation engine. An engineering effort in special-case handling of these and more views would increase hit rate. We plan to explore such optimizations in future. Finally, PUMA (and other Monkeys) and Brahmastra use fundamentally different techniques to navigate between app pages and it might be possible to combine them in a single system where PUMA is used if Brahmastra fails (or vice versa). In our experiments, such a hybrid approach would give an overall hit rate of 39% (total 397 apps).

[3] http://developer.android.com/guide/topics/ui/custom-components.html

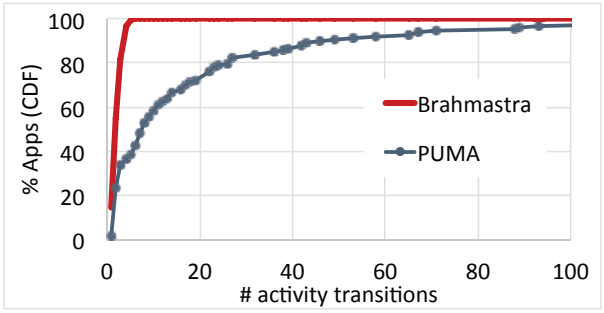


Figure 9: Test speed comparison of Brahmastra and PUMA

6.3 Speed

We use the number of activity transitions required to reach the target activity as a proxy for speed, since the actual time will vary depending on a variety of computational factors (e.g., network speed, device specifications). In Figure 9, we plot the CDF of the number of transitions required to reach the target activity for the apps which are successfully tested by both Brahmastra and PUMA. Since Brahmastra prunes away many unnecessary paths using static analysis, it runs faster than PUMA that suffers from uninformed activity transitions and large fanout in the activity transition graphs. On average, PUMA requires 18.7 transitions per app, while Brahmastra requires 2.5 transitions per app, resulting in 7 fold speedup.

7 Analysis of Ads in Kids Apps

Our first scenario is to use Brahmastra to study whether ad libraries for Android apps meet guidelines for protecting the online privacy of children. We give results for two popular ad components embedded in 220 kids apps. Our analysis shows that 80% of the apps displayed ads with a link to landing pages that have forms for collecting personal information, and 36% apps displayed ads deemed inappropriate to children.

7.1 Motivation and Goals

The Children's Online Privacy Protection Act (COPPA) lays down a variety of stipulations that mobile app developers must follow if their apps are directed at children under 13 years old [6]. In particular, COPPA disallows the collection of personal information by these apps unless the apps have first obtained parental consent.

COPPA holds the app developer responsible for the personal information collected by the embedded third party components as well as by the app's code [6]. Since it is common to see ad components included in free apps, we aim to measure the extent of potentially non-COPPA-compliant ad components in kids apps. Specifically, our first goal is to determine *whether in-app ads or landing pages pointed by these ads present forms that collect personal information*. Although displaying collection forms itself is not a violation, children might type in requested personal information, especially if these websites claim to offer free prizes or sweepstakes. In such cases, if these ads or landing pages do collect personal information without explicit parental consent, this act could be considered as a violation according to COPPA. Since it is difficult to model these legal terms into technical specifications, we only report potential concerns in this section. Our second goal is to test *whether content displayed in in-app ads or landing pages is appropriate for children*. Since this kind of judgement is fundamentally subjective, we show the breakdown of content categories as labeled by human testers.

Note that runtime observation is critical for this testing, since ads displayed within apps change dynamically depending on the inventory of ads at the time of request.

7.2 Testing Procedure

The testing has two steps. For a given app, we first collect ads displayed within apps and landing pages that are pointed by the ads. Second, for each ad and landing page, we determine: (1) whether they present forms to collect personal information such as first and last name, home address, and online contact as defined in COPPA; and (2) whether their content appears inappropriate to children and if so why.

Driving apps to display ads: We use Brahmastra to automatically drive apps to display ads. In this study, we focus on two popular ad libraries, AdMob and Millennial Media, because they account for over 40% of free Android apps with ads [11]. To get execution paths that produce ads, we use the following target methods as input to Brahmastra:

```
Lcom/google/ads/AdView;-><init>
Lcom/millennialmedia/android/MMAdView;-><init>
```

Collecting ads & landing pages: We redirect all the network traffic from executing test apps through a `Fiddler` proxy [8]. We install the `Fiddler` SSL certificate on the phone emulator as a trusted certificate to allow it to examine SSL traffic as well. We then identify the requests made by the ad libraries to their server component using domain names. Once these requests are collected, we replay these traces (several times, over several days), to fetch ad data from the ad servers as if these requests were made from these apps. This ad data is generally in the form of a JSON or XML object that contains details about the kind of ad served (image or text), the content to display on the screen (either text or the URL of an image), and the URL to redirect to if the ad is clicked upon. We record all of above for analysis.

Analyzing ads & landing pages: We use two methods to characterize ads and landing pages. First, for each landing page URL collected, we probe the Web of Trust (WoT) database [9] to get the "child safety" score. Second, to better understand the reasons why landing pages or ads may not be appropriate for children, we use crowdsourcing (via Amazon Mechanical Turk [3]) to label each ad and landing page and to collect detailed information such as the type of personal information that landing pages collect. As data collected from crowds may include inconsistent labeling, we use majority voting to filter out noise.

7.3 Results

Dataset: We collected our dataset in January 2014. To find apps intended for children, we use a list of apps categorized as "Kids" in Amazon's Android app store[4]. Since apps offered from the Amazon store are protected with DRM and resist bytecode rewriting, we crawled the Play store for apps with the same package name.

Starting from slightly over 4,000 apps in the Kids category, we found 699 free apps with a matching package name in the Play store. Among these, we find 242 apps that contain the AdMob or Millennial Media ad libraries. Using Brahmastra, we were successfully able to retrieve at least one ad request for 220 of these apps (also in January 2014), for which we report results in this section. For the remaining 22 apps, either Brahmastra could not navigate to the correct page, or the app did not serve any ad despite reaching the target page.

Results: We collected ads from each of the 220 apps over 5 days, giving us a total collection of 566 unique ads, and 3,633 unique landing pages. Using WoT, we determine that 183 out of the 3,633 unique landing pages have the child-safety score below 60, which fall in the "Unsatisfactory", "Poor" or "Very Poor" categories. 189 out of the 220 apps (86%) pointed to at least one of these pages during the monitoring period. Note that WoT did not contain child-safety ratings for 1,806 pages, so these

[4] Google Play store does not have a separate kids category.

Info Type	Landing Pages	Apps
Home address	47	58
First and last name	231	174
Online contact	100	94
Phone number	17	15
Total	235	175

Table 2: Personal information collected by landing pages

numbers represent a lower bound. We then used Amazon Mechanical Turk to characterize all 566 ads, and 2,111 randomly selected landing pages out of the 3,633. For each ad and landing page, we asked Amazon Mechanical Turk to check whether they collect personal information (of each type) and whether they contain inappropriate content for children (see Appendix B for the task details). We offered 7 cents (US) per each task (which involves answering various questions for each website or banner ad) and collected three responses per data point. As discussed above, we only counted responses that were consistent across at least two out of three respondents, to filter out noise.

Table 2 summarizes the types of personal information that landing pages ask users to provide as labeled by Amazon Mechanical Turk. We find that at least 80% of the apps in the dataset had displayed ads that point to landing pages with forms to collect personal information. On a manual examination of a subset of these pages, we found no labeling errors. We also found that none of the sites we manually checked attempt to acquire parental consent when collecting personal information. See Appendix B for examples.

Table 3 breaks down child-inappropriate content of the ads displayed in apps as labeled by Amazon Mechanical Turk. Although COPPA does not regulate the content of online services, we still find it concerning that 36% (80 out of 220) of the apps display ads with content deemed inappropriate for children. In particular 26% (58 apps) displayed ads that offer free prizes (e.g., Figure 13), which is considered a red flag of deceptive advertising, especially in ads targeting children as discussed in guidelines published by the Children's Advertising Review [10]. We also analysed the content displayed on the landing pages, and found a similar number of content violations as the ad images.

8 Analysis of Social Media Add-ons

Our second use case is to test apps against a recently discovered vulnerability associated with the Facebook SDK [30]. Our testing with Brahmastra shows that 13 out of 200 Android apps are vulnerable to the attack. Fixing it requires app developers to update the authentication logic in their servers as recommended by [30].

Content Type	Image Ads	Apps
Child exploitation	2	8
Gambling, contests, lotteries or sweepstakes	3	2
Misleading users about the product being advertised	7	16
Violence, weapons or gore	4	5
Alcohol, tobacco or drugs	3	3
Profanity and vulgarity	0	0
Free prize	39	58
Sexual or sexually suggestive content	12	29
Total	62	80

Table 3: Breakdown of child-inappropriate content in ads

8.1 Testing Goal

The Facebook access token vulnerability discussed in [30] can be exploited by attackers to steal the victim's sensitive information stored in vulnerable apps. For instance, if a malicious-yet-seemingly benign news app can trick the victim *once* to use the app to post a favorite news story on the victim's Facebook wall (which is a common feature found in many news apps), then the malicious app can use the access token obtained from the Facebook identity service to access sensitive information stored by *any* vulnerable apps that the victim had interacted with and have been associated with the victim's Facebook account. This attack can take place offline—once the malicious app obtains an access token, then it can send the token to a remote attacker who can impersonate as the victim to vulnerable apps.

Figure 10 gives the steps that allow a malicious application to steal `Victim`'s information from $\texttt{VulApp}_\text{s}$. The fact that the online service ($\texttt{VulApp}_\text{s}$) is able to retrieve the user's information from Facebook only means that the client ($\texttt{MalApp}_\text{c}$) possesses the privilege to the Facebook service, but is not a proof of the client app's identity ($\texttt{MalApp}_\text{c} \neq \texttt{VulApp}_\text{c}$). The shaded boxes in Figure 10 highlight the vulnerability. See [30] for more detail.

Wang et al. [30] manually tested 27 Windows 8 apps and showed that 78% of them are vulnerable to the access token attack. Our goal is to scale up the testing to a large number of Android applications. Note that testing for this vulnerability requires runtime analysis because the security violation assumptions are based on the interactions among the application, the application service, and Facebook service.

8.2 Testing Procedure

The testing has three steps. For a given app, we first need to drive apps to load a Facebook login screen. Second, we need to supply valid Facebook login credentials to observe interactions between the test application and Face-

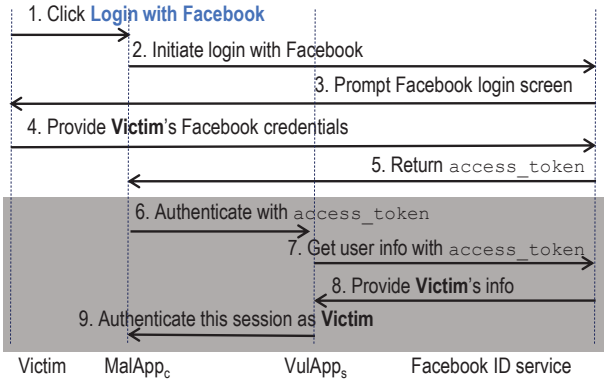


Figure 10: Facebook's access token, intended for authorizing access to `Victim`'s info, is used by `VulApp`$_s$ to authenticate the session as `Victim`. From step 9, `MalApp`$_c$ can steal `Victim`'s sensitive information in `VulApp`$_s$.

```
1  if(oSession.url.Contains("m.facebook.com")) {
2   var toReplace = "access_token=CAAHOi...";
3   ...
4   if(oSession.oResponse.headers.
5    ExistsAndContains("Location", "access_token"))
6   {
7    oSession.oResponse.headers["Location"] =
8     oSession.oResponse.headers["Location"].
9       replace(oRegEx, toReplace);
10   oSession["ui-customcolumn"] = "changed-header";
11  } }
```

Figure 11: A script used to manipulate `access_token`: We only show the first 6 bytes of the access token used in the attack.

book ID service. Third, we need to determine whether the test application misuses a Facebook access token for authenticating a client (steps 7-9) by monitoring network traffic and application behavior after providing a fraudulent access token.

Driving apps to display Facebook login: We use Brahmastra to automatically drive apps to invoke the Facebook SDK's authentication methods shown in Figure 7. Once the authentication methods open the sign-in window, we supply valid Facebook credentials.

Manipulating traffic with MITM proxy: As before, we direct all network traffic through a `Fiddler` proxy. Since Facebook sign-in traffic is encrypted over SSL, we also install a `Fiddler` SSL certificate on the phone emulator to decrypt all SSL traffic.

To manipulate the login, we record an `access_token` from a successful login session associated with another application (and therefore simulating an attacker as illustrated in the steps 1-5 of Fig. 10) and use the script shown in Fig. 11. It runs over HTTP responses, and overwrites an incoming `access_token` with a recorded one.

8.3 Experiments

Dataset: We randomly draw 200 apps from the dataset used in §6 for this testing.

Results: We find that 18 out of 200 apps use a Facebook access token for authentication, and among them 13 apps are vulnerable to a fraudulent access token (72%). 5 apps appear not vulnerable, and show some sort of error message when given a fraudulent access token. The remaining 182 apps use the Facebook SDK merely to post content to the user's wall, and not as an authentication mechanism. We determined this by looking at the network traffic at the login event, and observing that all of it is sent only to Facebook servers.

To understand how widespread the vulnerability is, we look at the statistics for the number of downloads on the Google Play store. Each of the 13 vulnerable apps has been downloaded more than 10,000 times, the median number of app downloads is over 500,000, and the most popular ones have been downloaded more than 10 million times. Further, these 13 apps have been built by 12 distinct publishers. This shows that the problem is not restricted to a few naïve developers. We shared the list of vulnerable apps with a Facebook security team on 2/27/2014 and got a response immediately that night that they had contacted the affected developers with the instructions to fix. The privacy implications of the possessing the vulnerability are also serious. To look at what user data can potentially be exfiltrated, we manually investigated the 13 vulnerable apps. Users of these apps may share a friends list, pictures, and messages (three dating apps); photos and videos (two apps); exercise logs and stats (one app); homework info (one app) or favorite news articles, books or music preferences (remaining six apps). By exploiting the vulnerability, a malicious app could exfiltrate this data.

9 Related Work

Automated Android app testing: A number of recent efforts proposed improvements over Android Monkey: AndroidRipper [13] uses a technique known as GUI ripping to create a GUI model of the application, and explores its state space. To improve code coverage, AndroidRipper relies on human testers to type in user credentials to get through blocking pages. However, despite this manual effort, the tool shows less than 40% code coverage after exploring an app for 4.5 hours. AppsPlayground [26] employs a number of heuristics—by guessing the right forms of input (e.g., email address, zip code) and by tracking widgets and windows in order to

reduce duplicate exploration. It shows that these heuristics are helpful although the coverage is still around 33%. SmartDroid [31] uses a combination of static and dynamic analysis to find the UI elements linked to sensitive APIs. However, unlike Brahmastra, SmartDroid explores every UI element at runtime to find the right view to click. A3E [14] also uses static analysis to find an activity transition graph and uses the graph to efficiently explore apps. We leveraged the proposed technique when building an execution path. However, similarly to the tools listed above, A3E again uses runtime GUI exploration to navigate through activities. In contrast to these works, Brahmastra determines an execution path using static analysis and rewrites an app to trigger a planned navigation, bypassing known difficulties related to GUI exploration.

Security analysis of in-app ads: Probably because only recently COPPA [6] had been updated to include mobile apps[5], we are not aware of any prior work looking into the issues around COPPA compliance of advertisements (and the corresponding landing pages) displayed within apps directed at children. However, several past works investigated security and privacy issues with respect to Android advertising libraries. AdRisk [22] is a static analysis tool to examine advertising libraries integrated with Android apps. They report that many ad libraries excessively collect privacy-sensitive information and expose some of the collected information to advertisers. Stevens et al. examine thirteen popular Android ad libraries and show the prevalent use of tracking identifiers and the collection of private user information [28]. Worse, through a longitudinal study, Book et al. show that the use of permissions by Android ad libraries has increased over the past years [18].

Analyzing logic flaws in web services and SDKs: The authentication vulnerability discussed in §8 falls into the category of logic flaws in web programming. Recent papers have proposed several technologies for testing various types of logic flaws [16, 17, 21]. However, these techniques mainly target logic flaws in *two-party web programs*, i.e., programs consisting of a client and a server. Logic flaws become more complicated and intriguing in multi-party web programs, in which a client communicating with multiple servers to accomplish a task, such as the Facebook-based authentication that we focus in this paper. AuthScan is a recently developed technique to automatically extract protocol specifications from concrete website implementations, and thus discover new vulnerabilities in the websites [15]. In contrast, our goal is not to discover any new vulnerability on a website, but to scale up the testing of a known vulnerability to a large number of apps.

[5]The revision was published on July 2013.

10 Discussion

Limitations: Although Brahmastra improves test hit rates over Monkey-like tools, we discover several idiosyncratic behaviors of mobile apps that challenge runtime testing. Some apps check servers upon launching and force upgrading if newer versions exist. Some apps constantly load content from remote servers, showing transient behaviors (e.g., extremely slow at times). We also have yet to implement adding callbacks related to sensor inputs. Another challenge is to isolate dependent components in the code. We assume that each activity is more or less independent (except that they pass parameters along with intent) and use our jump start technique to bypass blocking pages and to speed up testing. However, we leave as future work a task to statically determine dependent activities to find activities to jump-start to without affecting the program behavior.

Other runtime security testing of mobile apps: As mobile apps are highly driven by user interaction with visual components in the program, it is important to analyze the code behavior in conjunction with runtime UI states. For instance, malicious third-party components can trick users into authorizing the components to access content (e.g., photos) that the users intended to share with the application. Brahmastra can be used to capture visual elements when certain APIs are invoked to check against such click jacking attempts. Brahmastra can also automate the testing to check whether privacy-sensitive APIs are only invoked with explicit user interactions.

11 Conclusion

We have presented a mobile app automation tool, Brahmastra, that app store operators and security researchers can use to test third-party components at runtime as they are used by real applications. To overcome the known shortcomings of GUI exploration techniques, we analyze application structure to discover desired execution paths. Then we re-write test apps to follow a short path that invokes the target third-party component. We find that we can more than double the test hit rate while speeding up testing by a factor of seven compared to a state-of-the-art Monkey tool.

We use Brahmastra for two case studies, each of which contributes new results: checking if third-party ad components in kids apps are compliant with child-safety regulations; and checking whether apps that use Facebook Login are vulnerable to a known security flaw. Among the kids apps, we discover 36% of 220 kids apps display ads deemed inappropriate for children, and 80% of the apps display ads that point to landing pages which attempt to collect personal information without parental consent. Among the apps that use Facebook Login, we find that 13 applications are still vulnerable to the Face-

book access token attack even though the attack has been known for almost a year. Brahmastra let us quickly check the behavior of hundreds of apps for these studies, and it can easily be used for other studies in the future—checking whether privacy-sensitive APIs can be invoked without explicit user interaction, discovering visible UI elements implicated in click jacking attempts, and more.

Acknowledgments

This material is based on research sponsored in part by DARPA under agreement number FA8750-12-2-0107, NSF CCF-1139021, and University of Maryland Partnership with the Laboratory of Telecommunications Sciences, Contract Number H9823013D00560002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A tool for reverse engineering Android apk files. http://code.google.com/p/android-apktool/.

[2] Activity — Android Developers. http://developer.android.com/reference/android/app/Activity.html.

[3] Amazon Mechanical Turk. https://www.mturk.com.

[4] Android Debug Bridge. http://developer.android.com/tools/help/adb.html.

[5] Android Developers, The Developer's Guide. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[6] Complying with COPPA: Frequently Asked Questions. http://business.ftc.gov/documents/Complying-with-COPPA-Frequently-Asked-Questions.

[7] Dexpler: A Dalvik to Soot Jimple Translator. http://www.abartel.net/dexpler/.

[8] Fiddler. http://www.telerik.com/fiddler.

[9] Web of Trust. https://www.mywot.com/.

[10] Self-Regulatory Program for Childrens Advertising, 2009. http://www.caru.org/guidelines/guidelines.pdf.

[11] AppBrain, Feb. 2014. http://www.appbrain.com/stats/libraries/ad.

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[13] D. Amalfitano, A. R. Fasolino, S. D. Carmine, A. Memon, and P. Tramontana. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the IEEE Conference on Automated Software Engineering (ASE)*, 2012.

[14] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, 2013.

[15] G. Bai, J. Lei, G. Meng, S. S. V. P. Saxena, J. Sun, Y. Liu, and J. S. Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *NDSS*, 2013.

[16] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: Automatically detecting parameter tampering vulnerabilities in web applications. In *CCS*, 2010.

[17] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In *CCS*, 2011.

[18] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. In *IEEE Mobile Security Technologies (MoST)*, 2013.

[19] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[20] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *CCS*, 2013.

[21] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security*, 2010.

[22] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *WiSec*, 2012.

[23] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *Mobisys*, 2014.

[24] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[25] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *USENIX NSDI*, 2014.

[26] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 2013.

[27] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *Mobisys*, 2014.

[28] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *IEEE Mobile Security Technologies (MoST)*, 2012.

[29] R. Valle-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *IBM Centre for Advanced Studies Conference*, 1999.

[30] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security*, 2013.

[31] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

A Constant Propagation Analysis

We extend the existing constant propagation analysis so as to trace `intents`, UI elements, and listeners. In addition to traditional value types, such as numerical or string constant, we add meta-class, object, and intent sorts, which track class ids, object references, and intent instances, respectively. For instructions that create objects; load class ids; or invoke special APIs such as `Intent.setClass()`, we add their semantics into the data-flow transfer function.

Figure 12 illustrates how we extend data-flow lattice; how we conform to *meet* operation property; and how we define semantics of relevant instructions.

```
type lattice = ...
  | Clazz of string (* const−class *)
  | Object of string (* instance *)
  | Intent of string (* Intent for a specific component *)
  | ...
let meet l1 l2 = match l1, l2 with ...
  | Clazz c1, Clazz c2 when 0 = compare c1 c2 → l1
  | Object o1, Object o2 when 0 = compare o1 o2 → l1
  | Intent i1, Intent i2 when 0 = compare i1 i2 → l1
  | ...
let transfer (inn: lattice Map.t) (op, opr) = ...
  else if OP_NEW = op then (* NEW *)
  (
    let dst :: id ::[] = opr in
    let cname = Dex.get_ty_name id in
    if 0 = compare cname '' Intent ''
    then Map.add dst (Intent ``'') inn
    else Map.add dst (Object cname) inn
  ) ...
```

Figure 12: Abbreviated source code of extended constant propagation analysis. Meta-class, object, and intent sorts maintain information as string, and they can be merged only if internal values are identical, hence *must*-analysis. As an example, this shows how to handle opcode `NEW`.

B Examples of Ads in Kids Apps

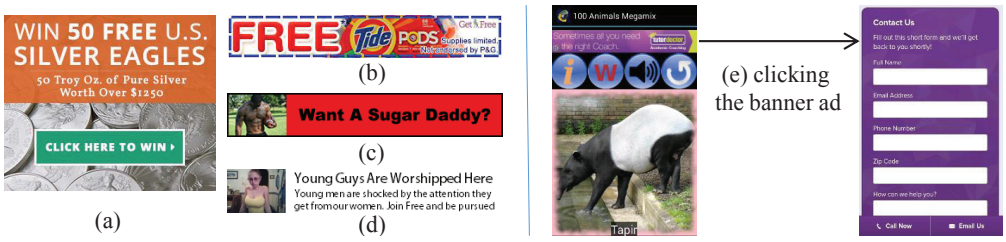


Figure 13: a) and (b) offer a free prize and (c) and (d) are sexually suggestive. (e) shows an example where clicking a banner ad displayed in a kids app opens up a landing page that presents forms to collect personal information.

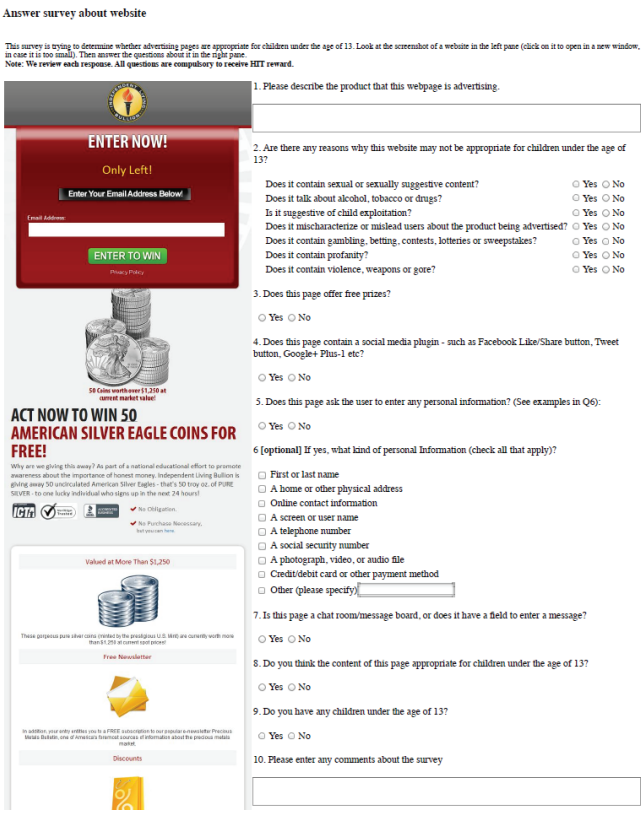


Figure 14: A screenshot of the Amazon Mechanical Turk task that we created to characterize landing pages pointed by ads displayed in kids apps

Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks

Qi Alfred Chen, Zhiyun Qian†, Z. Morley Mao
University of Michigan, †NEC Labs America, Inc.
alfchen@umich.edu, zhiyunq@nec-labs.com, zmao@umich.edu

Abstract

The security of smartphone GUI frameworks remains an important yet under-scrutinized topic. In this paper, we report that on the Android system (and likely other OSes), a weaker form of GUI confidentiality can be breached in the form of UI state (not the pixels) by a background app without requiring any permissions. Our finding leads to a class of attacks which we name *UI state inference attack*. The underlying problem is that popular GUI frameworks by design can potentially reveal every UI state change through a newly-discovered public side channel — shared memory. In our evaluation, we show that for 6 out of 7 popular Android apps, the UI state inference accuracies are 80–90% for the first candidate UI states, and over 93% for the top 3 candidates.

Even though the UI state does not reveal the exact pixels, we show that it can serve as a powerful building block to enable more serious attacks. To demonstrate this, we design and fully implement several new attacks based on the UI state inference attack, including hijacking the UI state to steal sensitive user input (*e.g.,* login credentials) and obtain sensitive camera images shot by the user (*e.g.,* personal check photos for banking apps). We also discuss non-trivial challenges in eliminating the identified side channel, and suggest more secure alternative system designs.

1 Introduction

The confidentiality and integrity of applications' GUI content are well recognized to be critical in achieving end-to-end security [1–4]. For instance, in the desktop and browser environment, window/UI spoofing (*e.g.,* fake password dialogs) breaks GUI integrity [3, 4]. On the Android platform, malware that takes screenshots breaches GUI confidentiality [5]. Such security issues can typically lead to the direct compromise of the confidentiality of user input (*e.g.,* keystrokes). However, a weaker form of confidentiality breach has not been thoroughly explored, namely the knowledge of the application UI state (*e.g.,* knowing that the application is showing a login window) *without knowing the exact pixels of the screen*, especially in a smartphone environment.

Surprisingly, in this paper we report that on the Android system (and likely on other OSes), such GUI confidentiality breach is indeed possible, leading to serious security consequences. Specifically, we show that *UI state* can be inferred without requiring any Android permissions. Here, *UI state* is defined as a mostly consistent user interface shown in the window level, reflecting a specific piece of program functionality. An example of a UI state is a login window, in which the text content may change but the overall layout and functionality remain the same. Thus, we call our attack *UI state inference attack*. In this attack, an attacker first builds a UI state machine based on UI state signatures constructed offline, and then infers UI states in real time from an unprivileged background app. In Android terminology, the UI state is known as *Activity*, so we also call it *Activity inference attack* in this paper.

Although UI state knowledge does not directly reveal user input, due to a lack of direct access to the exact pixels or screenshots, we find that it can effectively serve as a building block and enable more serious attacks such as stealing sensitive user input. For example, based on the inferred UI states, we can further break the GUI integrity by carefully exploiting the designed functionality that allows UI preemption, which is commonly used by alarm or reminder apps on Android.

The fundamental reason for such confidentiality breach is in the Android GUI framework design, where every UI state change can be unexpectedly observed through publicly accessible side channels. Specifically, the major enabling factor is a newly-discovered *shared-memory side channel*, which can be used to detect window events in the target application. This side channel exists because shared memory is commonly adopted by window managers to efficiently receive window changes or updates from running applications. For more details, please refer to §2.1 where we summarize the design and implementation of common window managers, and §3.2 where we describe how shared memory plays a critical role. In fact, this design is not specific to Android: nearly all popular OSes such as Mac OS X, iOS, and Windows also adopt this shared-memory mechanism for their win-

dow managers. Thus, we believe that our attack on Android is likely to be generalizable to other platforms.

Since the window manager property we exploit has no obvious vulnerabilities in either design or implementation, it is non-trivial to construct defense solutions. In §9, we discuss ways to eliminate the identified side channels, and also suggest more secure alternative system designs.

Our discovered Activity inference attack enables a number of serious follow-up attacks including (1) *Activity hijacking attack* that can unnoticeably hijack the UI state to steal sensitive user input (*e.g.,* login credentials), and (2) *camera peeking attack* that captures sensitive camera images shot by the user (*e.g.,* personal check photos for banking apps). We have fully designed and implemented these attacks and strongly encourage readers to view several short video demos at **`https://sites.google.com/site/uistateinferenceattack/demos`** [6].

Furthermore, we demonstrate other less severe but also interesting security consequences: (1) existing attacks [5, 7–10] can be enhanced in stealthiness and effectiveness by providing the target UI states; (2) user behavior can be inferred through tracking UI state changes. Previous work has demonstrated other interesting Android side-channel attacks, such as inferring the web pages a user visits [11] as well as the identity, location, and disease information of users [12]. However, these attacks are mostly application-specific with limited scope. For instance, Memento [11] only applies to web browsers, and Zhou *et al.* [12] reported three side channels specific to three different apps. In contrast, the UI state concept in this paper applies generally to all Android apps, leading to not only a larger attack coverage but also many more serious attacks, such as the Activity hijacking attack which violates GUI integrity.

The contributions of this paper are as follows:

- We formulate the general UI state inference attack that violates a weaker form of GUI confidentiality, aimed at exposing the running UI states of an application. It exploits the unexpected interaction between the design and implementation of the GUI framework (mainly the window manager) and a newly-discovered shared-memory side channel.
- We design and implement the Android version of this attack and find an accuracy of 80–90% in determining the foreground Activity for 6 out of 7 popular apps. The inference itself does not require any permissions.
- We develop several attack scenarios using the UI state inference technique and demonstrate that an attacker can steal sensitive user input and sensitive camera images shot by the user when using Android apps.

For the rest of the paper, we first provide the attack background and overview in §2. The newly-discovered side channel and Activity transition detection are detailed in §3, and based on that, the Activity inference technique is described in §4. In §5, we evaluate this attack with popular apps, and §6, §7 and §8 show concrete follow-up attacks. We cover defense in §9, followed by related work in §10, before concluding in §11.

2 Background and Overview

2.1 Background: Window Manager

Window manager is system software that interacts with applications to draw the final pixels from all application windows to the frame buffer, which is then displayed on screen. After evolving for decades, the most recent design is called *compositing window manager*, which is used virtually in all modern OSes. Unlike its predecessors, which allow individual applications to draw to the frame buffer directly, a compositing window manager requires applications to draw the window content to *off-screen buffers* first, and use a dedicated window compositor process to combine them into a final image, which is then drawn to the frame buffer.

Client-drawn and server-drawn buffer design. There are two types of compositing window manager design, as shown in Fig. 1. In this figure, *client* and *server* refer to an application and the window compositor[1] respectively. In the client-drawn buffer design, the applications draw window content to off-screen buffers, and use IPC to communicate these buffers with the server, where the final image is composited and written to the frame buffer. This design is very popular and is used in Mac OS X, iOS, Windows, Android, and Wayland for the future Linux [13, 14]. In the server-drawn buffer design, the main difference is that the off-screen buffers are allocated and drawn by the window compositor instead of by the applications. Applications send commands to the window compositor to direct the drawing process. Only the X window system on the traditional Linux and Mir for the future Linux [15] use this design.

Both designs have their advantages. The client-drawn buffer design provides better isolation between applications, more flexible window drawing and more balanced overhead between the client and the server. For the server-drawn buffer design, the server has control over all applications' window buffers, which is better for centralized resource management. Interestingly, some prior work choose the former to enhance GUI security [1], but we find that it actually enables our attacks (shown in §3).

2.2 Background: Android Activity and Activity Transition

In Android, the UI state our attack infers is called *Activity*. An Activity provides a user interface (UI) for user in-

[1] For traditional Linux the server is an X server, and the window compositor is a separate process talking to the X server. In Fig. 1 we refer to the combination of them as the window compositor.

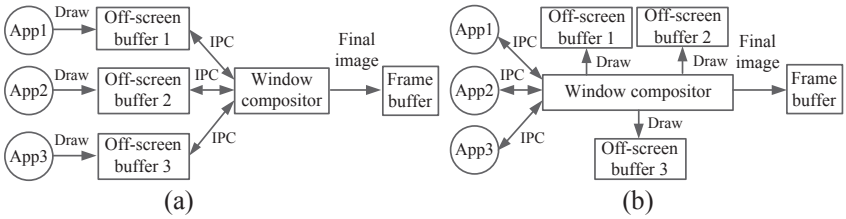


Figure 1: Two types of compositing window manager design: (a) client-drawn buffer design, and (b) server-drawn buffer design. *Client* refers to the application, and *server* refers to the window compositor.

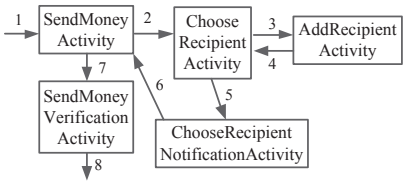


Figure 2: Activities involved in sending money in Android Chase app. The numbers denote the action order.

teractions, and is typically a full-screen window serving as a functionality unit in Android. We denote Activities as $a, b, ...$, and the set of Activities for an app as A. Due to security concerns, by default apps cannot know which Activity is currently shown in the foreground unless they are the owners or the central Activity manager.

An Activity may display different content depending on the app state. For instance, a dictionary app may have a single "definition" Activity showing different texts for each word lookup. We call these distinct displays *ViewStates*, and denote the set of them for Activity a as $a.VS$.

Activity transition. In Android, multiple Activities typically work together and transition from one to another to support the functionality of an app as a whole. An example is shown in Fig. 2. During a typical transition, the current foreground Activity pauses and a new one is created. A Back Stack [16] storing the current and past Activities is maintained by Android. To prevent excessive memory usage, at any point in time, only the top Activity has its window buffer allocated. Whenever an Activity transition occurs, the off-screen buffer allocation for the new Activity window and the deallocation for the existing Activity window take place.

Activity transitions can occur in two ways: a new Activity is created (*create* transition), or an existing one resumes when the BACK key is pressed (*resume* transition), corresponding to push and pop actions in the Back Stack. Fig. 3 shows the major function calls involved in these two transition types. Both transition types start by pausing the current foreground Activity, and then launching the new one. During launch, the *create* transition calls both `onCreate()` and `onResume()`, while the *resume* transition only calls `onResume()`. Both `onCreate()` and `onResume()` are implemented by the app. After that, `performTraversal()` is called, in which `measure()` and `layout()` calculate the sizes and locations of UI components, and `draw()` puts them into a data structure as the new Activity UI. Finally, the *create* transition pushes the new Activity into the Back Stack and stops the current one, while the *resume* transition pops the current one and destroys it.

Activity transition graph. Immediately after a transition, the user lands on one of the ViewStates of the new Activity, which we call a *LandingState*. We denote the set of LandingStates for Activity a as $a.LS$, and $a.LS \subseteq a.VS$. Individual LandingStates are denoted as $a.ls_1, a.ls_2, ...$ Activity transition is a relationship $a.VS \rightarrow b.LS, a, b \in A$. As the ViewState before the transition is not of interest in this study, we simplify it to $a \rightarrow b.LS$, which forms the graph in Fig. 4. Note that our definition is slightly different from that in previous work [17] as the edge tails in our graph are more fine-grained: they are LandingStates instead of Activities.

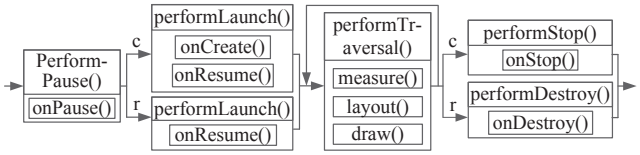


Figure 3: The function call trace for *create* (denoted by c) and *resume* (denoted by r) transitions.

2.3 Attack Overview

Our proposed UI state inference is a general side-channel attack against GUI systems, aimed at exposing the running UI state of an application at the window level, *i.e.,* the currently displayed window (without knowing the exact pixels). To achieve that, the attack exploits a newly-discovered shared-memory side channel, which may exist in nearly all popular GUI systems used today (shown in §3). In this paper, we focus on the attack on the Android platform: Activity inference attack. We expect the technique to be generalizable to all GUI systems with the same window manger design as that in Android, such as the GUI systems in Mac OS X, iOS, Windows, *etc.*

Threat model. We require an attack app running in the background on the victim device, which is a common requirement for Android-based attacks [7–11, 18]. To ensure stealthiness, the app should be low-overhead, not draining the battery too quickly. Also, as the purpose of permissions is to alert users to privacy- or security-invasive apps [19], the attack should not require any additional permissions besides a few very common ones, for example the INTERNET permission.

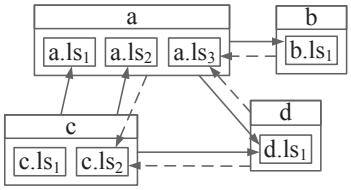


Figure 4: An example Activity transition graph. Solid and dotted edges denote *create* and *resume* transitions respectively.

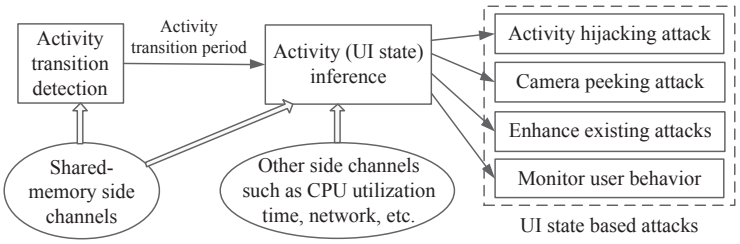


Figure 5: Activity inference attack overview.

General steps. As shown in Fig. 5, Activity inference is performed in two steps:

1. Activity transition detection: we first detect an Activity transition event, which reports a single bit of information on whether an Activity transition just occurred. This is enabled by the newly-discovered shared-memory side channel. As shown later in §3.3, the change observed through this channel is a highly-distinct "signal".

2. Activity inference: upon detecting an Activity transition, we need to differentiate which Activity is entering the foreground. To do so, we design techniques to train the "signature" for the landing Activity, which roughly characterizes its starting behavior through publicly observable channels, including the new shared-memory side channel, CPU utilization time, and network activity (described in §4).

Finally, using our knowledge of the foreground Activity in real time, we develop novel attacks that can effectively steal sensitive user input as well as other information as detailed in §6, §7 and §8.

3 Shared-Memory Side Channel and Activity Transition Detection

In this section, we first report the newly-discovered side channel along with the fundamental reason for its existence, and then detail the transition detection technique.

3.1 Shared-Memory Side Channels

As with any modern OS design, the memory space of an Android app process consists of the private space and the shared space. Table 1 lists memory counters in */proc/pid/statm* and their names used in the Linux command *top* and the Linux source code. Inherited from Linux, these counters can be freely accessed without any privileges. With these counters, we can calculate both the private and shared sizes for virtual memory and physical memory. In this paper, we leverage `mm->shared_vm` and `file_rss` as our shared-memory side channels, the former for virtual memory and the latter for physical memory. For convenience, we refer to them as *shared_vm* and *shared_pm*. In this section, we focus on using *shared_vm* to detect Activity transition events. In §4.1, we use both *shared_vm* and *shared_pm* to infer Android Content Provider usages in the Activity inference, which is another use case we discovered.

3.2 Android Window Events and Shared-Memory Side Channel

We find that *shared_vm* changes are correlated with Android window events. In this section, we detail its root cause and prevalence in popular GUI systems.

Shared-memory IPC used in the Android window manager. As mentioned earlier in §2.1, Android adopts the client-drawn buffer design, where each client (app) needs to use a selected IPC mechanism to communicate their off-screen buffers with the window compositor. In practice, we find that shared memory is often used, since it is the most efficient channel (without large memory copy overhead). On Android, when an Activity transition occurs, *shared_vm* size changes can be found in both the app process and the window compositor process named *SurfaceFlinger*. More investigations into Android source code reveal that the size changes correspond to the allocations and deallocations of a data structure named *GraphicBuffer*, which is the off-screen buffer in Android. In the Android window drawing process shown in Fig. 3, GraphicBuffer is shared between the app process and the SurfaceFlinger process using `mmap()` at the beginning of `draw()` in `performTraversal()`.

Interestingly, this implies that if we know the GraphicBuffer size for a target window, we can detect its allocation and deallocation events by monitoring the size changes of *shared_vm*. Since the GraphicBuffer size is proportional to the window size, and an Activity is a full-screen window, its GraphicBuffer size is fixed for a given device, which can be known beforehand.

It is noteworthy that different from private memory space, shared memory space changes only when shared files or libraries are mapped into the virtual memory. This keeps our side channel clean; as a result, the changes in *shared_vm* are distinct with minimum noise.

Shared-memory side-channel vulnerability on other OSes. To understand the scope, we investigate other OSes besides Android. On Linux, Wayland makes it clear that it uses shared buffers as IPC between the win-

Item in */proc/-pid/statm*	Description	Name in *top*	Name in Linux source code
VmSize	Total virtual memory size	VIRT	`mm->total_vm`
drs	Private virtual memory size	/	`mm->total_vm-mm->shared_vm`
resident	Total physical memory size	RES	`file_rss+anon_rss`
share	Shared physical memory size	SHR	`file_rss`

Table 1: Android/Linux memory counters in */proc/pid/statm* and their names in the Linux command *top*, and the Linux source code (obtained from `task_statm()` in `task_mmu.c`). The type of `mm` is `mm_struct`.

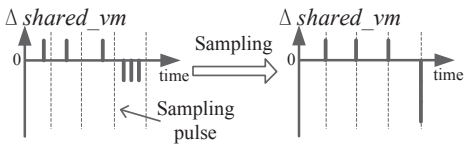


Figure 6: A successful sampling of an Activity transition signal for the Activity transition detection

dow compositor and clients to render the windows [13]. Similar to Android, attackers can use */proc/pid/statm* to get the shared memory size and detect window events.

Mac OS X, iOS and Windows neither explain this IPC in their documentations nor have corresponding source code for us to explore, so we did some reverse engineering using memory analysis tools such as VMMap [20]. On Windows 7, we found that whenever we open and close a window, a memory block appears and disappears in the shared virtual memory space of both the window compositor process, *Desktop Window Manager*, and the application process. Moreover, the size of this memory block is proportional to the window size on the screen. This strongly implies that this memory block is the off-screen buffer and shared memory is the IPC used for passing it to the window compositor. Thus, using the `GetProcessMemoryInfo()` API that does not require privilege, similar inference may be possible.

Mac OS X is similar to Windows except that the memory block in shared memory space is named *CG backing store*. On iOS this should be the same as Mac OS X since they share the same window compositor, *Quartz Compositor*. But on Mac OS X and iOS, only system-wide aggregated memory statistics seem available through `host_statistics()` API, which may still be usable for this attack but with a less accuracy.

3.3 Activity Transition Detection

With the above knowledge, detecting Activity transition is simply a matter of detecting the corresponding window event pattern by monitoring *shared_vm* size changes.

The left half of Fig. 6 shows the typical *shared_vm* changing pattern for an Activity transition, and we name it *Activity transition signal*. In this signal, the positive and negative spikes are increases and decreases in *shared_vm* respectively, corresponding to GraphicBuffer allocations and deallocations. The GraphicBuffer allocation for the new Activity usually happens before the deallocation for the current Activity, which avoids user-visible UI delays. Since Activity windows all have full-screen sizes, the increase and decrease amount are the same. With the multiple buffer mechanism for UI drawing acceleration on Android [21], 1–3 GraphicBuffer allocations or deallocations can be observed during a single transition, resulting in multiple spikes in Fig. 6. The delay between allocations is usually 100–500 ms due to measurement and layout computations, while the delay between deallocations is usually under 10 ms. An example result of a successful sampling is shown on the right half of Fig. 6 with the sampling period being 30–100 ms.

To detect this signal, we monitor the changes of *shared_vm*, and conclude an *Activity transition period* by observing (1) both full-screen size *shared_vm* increase and decrease events, (2) the idle time between two successive events is longer than a threshold *idle_thres*. A successful detection is shown on the top of Fig. 10.

We evaluate this method and find a very low false positive rate, which is mainly because the *shared_vm* channel is clean. In addition, it is rare that the following unique patterns happen randomly in practice: (1) the *shared_vm* increase and decrease amounts are exactly the same as the full-screen GraphicBuffer size (920 pages for Samsung Galaxy S3); (2) both the increase and decrease events occur very closely in time.

On the other hand, this method may have false negatives due to a cancellation effect — when an increase and a decrease are in the same sampling period, they cancel each other and the *shared_vm* size stays unchanged. Raising the sampling rate can overcome this problem, but at the cost of increased sampling overhead. Instead, we solve the problem using the number of minor page faults (*minflt*), in */proc/pid/stat*. When allocating memory for a GraphicBuffer, the physical memory is not actually allocated until it is used. At time of use, the same number of pages faults as the allocated GraphicBuffer page size is generated. Since *minflt* monotonically increases as a cumulative counter, we can use it to deduce the occurrence of a cancellation event.

4 Activity Inference

After detecting an Activity transition, we infer the identity of the new Activity using two kinds of information:

1. Activity signature. Among functions involved in the transition (as shown in Fig. 3), `onCreate()` and `onResume()` are defined in the landing Activity, and the execution of `performTraversal()` depends on the UI elements and layout in its LandingState. Thus, every transition has behavior specific to the landing Activ-

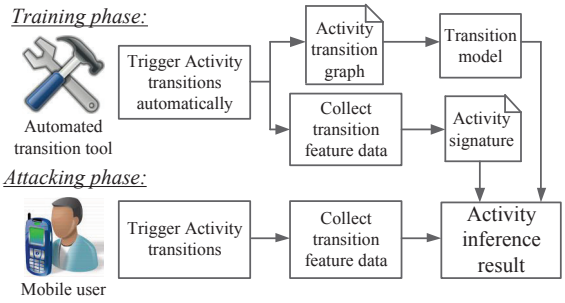


Figure 7: Overview of the Activity inference

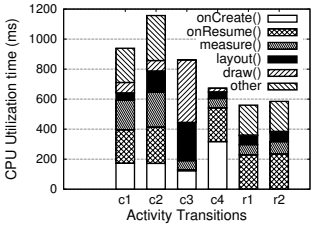


Figure 8: CPU utilization time breakdown for 6 Activity transitions in WebMD

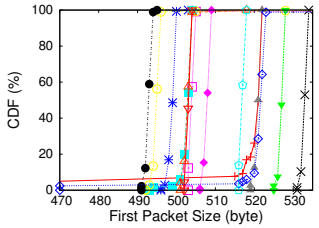


Figure 9: CDF of the first packet sizes for various LandingStates in H&R Block

ity, giving us opportunities to infer the landing Activity based on feature data collected during the transition.

2. Activity transition graph. If the Activity transition graph of an app is sparse, once the foreground Activity is known, the set of the next candidate Activities is limited, which can ease the inference difficulty. Thus, we also consider Activity transition graph in the inference.

Fig. 7 shows an overview of the Activity inference process. This process has two phases, the training phase and the attacking phase. The training phase is executed first offline. We build a tool to automatically generate Activity transitions in an app, and at the same time collect feature data to build the Activity signature and construct the Activity transition graph. In the attacking phase, when a user triggers an Activity transition event, the attack app first collects feature data like in the training phase, then leverages Activity signature and a transition model based on the Activity transition graph to perform inference.

4.1 Activity Signature Design

During the transition, we identify four types of features described below and use them jointly as the signature.

Input method events. Soft keyboard on smartphones is commonly used to support typing in Activities. It usually pops up automatically at the landing time. There is also a window event for the keyboard process, which again can be inferred through *shared_vm*. This is a binary feature indicating whether the LandingState requires typing.

Content Provider events. Android component Content Provider manages access to a structured set of data using SQLite as backend. To deliver content without memory copy overhead, Android uses anonymous shared memory (*ashmem*) shared by the Content Provider and the app process. Similar to the compositing window manger design, by monitoring *shared_vm*, we can detect the query and release events of the Content Provider. Specifically, in Android design, we found that the virtual memory size of ashmem allocated for a Content Provider query is a fixed large value, *e.g.,* 2048 KB for Android 4.1, which creates a clear signal. Usually its content only constitutes a small portion. To know the content size, we also monitor *shared_pm* introduced in §3.1, which indicates the physical memory allocation size for the content.

The Content Provider is queried in `onCreate()` and `onResume()` to show content in the landing Activity. For signature construction, we collect Content Provider query events and the corresponding content size by monitoring *shared_vm* and *shared_pm*. As *shared_pm* may be noisy, we use a normal distribution to model the size.

CPU utilization time. Fig. 8 shows the CPU utilization time collected by DDMS [22] for each function in Fig. 3 during the transition. For the 6 transitions, c and r denote *create* and *resume* transition, and 1–4 denote 4 different LandingStates. The time collected may be inflated due to the overhead added by DDMS profiling. The figure shows that CPU utilization time spent in each function differs across distinct LandingStates due to distinct drawing complexity, and for the same LandingState, *resume* transitions usually take less time than *create* ones since the former lacks `onCreate()`. Thus, the CPU utilization time can be used to distinguish Activity transitions with different transition types and LandingStates.

To collect data for this feature, we record the user space CPU utilization time value (*utime*), in */proc/pid/stat* for the Activity transition. Similar to previous work [23, 24], we find that the value for the same repeated transition roughly follows normal distribution. Thus, we model it using a normal distribution.

Network events. For LandingStates with content from the network, network connection(s) are initiated in `performLaunch()` during the transition. For a given LandingState, the request command string such as HTTP GET is usually hard-coded, with only minor changes from app states or user input. This leads to similar size of the first packet immediately after the connection establishment. We do not use the response packet size, since the result may be dynamically generated. Fig. 9 shows the CDF of the first packet sizes for 14 Activity LandingStates in H&R Block. As shown, most distributions are quite stable, with variations of less than 3 bytes.

To capture the first packet size, we monitor the send packet size value in */proc/uid_stat/uid/tcp_snd*. We con-

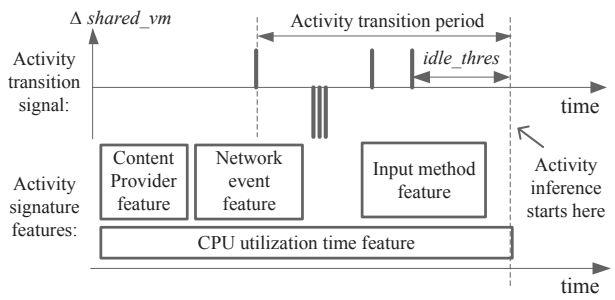


Figure 10: Signature feature data collection timeline.

currently monitor */proc/net/tcp6*, which contains network connection information such as the destination IP address and app uid for all apps. We use the uid to find the app which the connection belongs to, and use time correlation to match the first packet size to the connection. For the LandingState with multiple connections, we use the *whois* IP address registry database [25] to get the organization names and thus distinguish connections with different purpose. To read first packet sizes accurately, we raise the sampling rate to be 1 in 5 ms during the transition period. Since this period usually lasts less than 1 second, the overall sampling overhead is still low.

For signature construction, we keep separate records for connections with different organization names and occurrence ordering. For each record, we use the first packet size appearance frequencies to calculate the probability for this feature.

Fig. 10 shows the data collection timeline for these feature data and their relationship with the *shared_vm* signal. The Content Provider event feature is collected before the first *shared_vm* increase, and the input method event feature is collected after the first *shared_vm* decrease. Network events are initiated before the first *shared_vm* increase, while the first packet size is usually captured after that. The CPU utilization time feature is collected throughout the whole transition period.

With these four types of features, our signature probability $Prob(\langle\cdot, a.ls_i\rangle)$, $a \in A$, $a.ls_i \in a.LS$ is obtained by computing the product of each feature's probability, based on the observation that they are fairly independent. In §5, we evaluate our signature design with these four features both jointly and separately.

4.2 Transition Model and Inference Result

Transition model. In our inference, the states (*i.e.,* Activities) are not visible, so we use Hidden Markov Model (HMM) to model Activity transitions. We denote the foreground Activity trace with n Activity transitions as $\{a_0, a_1, ..., a_n\}$. The HMM can be solved using the Viterbi algorithm [26] with initialization $Prob(\{a_0\}) = \frac{1}{|A|}$, and inductive steps $Prob(\{a_0, ..., a_n\}) = \underset{a_n.ls_i \in a_n.LS}{\arg\max} Prob(\langle\cdot, a_n.ls_i\rangle) Prob(a_n|a_{n-1}) Prob(\{a_0, ..., a_{n-1}\})$.

In inductive steps, $Prob(\langle\cdot, a_n.ls_i\rangle)$ denotes the probability calculated from Activity signature, and $Prob(a_n|a_{n-1})$ denotes the probability that a_{n-1} transitions to a_n. If a_{n-1} has x egress edges in the transition graph, $Prob(a_n|a_{n-1}) = \frac{1}{x}$, assuming that user choices are uniformly distributed.

The typical Viterbi algorithm [26] calculates the most likely Activity trace $\{a_0, a_1, ..., a_n\}$, with computation complexity $O((n+1)|A|^2)$. However, for our case, only the new foreground Activity a_n is of interest, so we modify the Viterbi algorithm by only calculating $Prob(\{a_{n-c+1}, ..., a_n\})$, where c is a constant. This reduces the computation complexity to $O(c|A|^2)$. In our implementation, we choose $c = 2$.

Inference result. After inference, our attack outputs a list of Activities in decreasing order of their probabilities.

4.3 Automated Activity Transition Tool

By design, both the Activity signature and Activity transition graph are mostly independent of user behavior; therefore, the training phase does not need any victim participation. Thus, we also develop an automated tool to accelerate the training process offline.

Implementation. Our tool is built on top of `ActivityInstrumentationTestCase`, an Android class for building test cases of an app [27]. The implementation has around 4000 lines of Java code.

Activity transition graph generation with depth-first search. To generate the transition graph, we send and record user input events to Activities to drive the app in a depth-first search (DFS) fashion like the depth-first exploration described in [17]. The DFS graph has ViewStates as nodes, user input event traces as edges (*create* transitions), and the BACK key as the back edge (*resume* transitions). Once the foreground Activity changes, transition information such as the user input trace and the landing Activity name is recorded. The graph generated is in the form of the example shown in Fig. 4.

Activity transition graph traversal. With the transition graph generated, our tool supports automatic graph traversals in deterministic and random modes. In the random mode, the tool chooses random egress edges during the traversal, and goes back with some probabilities.

Tool limitations. We assume Activities are independent from each other. If changes in one Activity affect Activity transition behavior in another, our tool may not be aware of that, leading to missed transition edges. For some user input such as text entering, the input generation is only a heuristic and cannot ensure 100% coverage. To address such limitations, some human effort is involved to ensure that we do not miss the important Activities and ViewStates.

Application name	Activity number	Activity transitions					Activity LandingStates			
		create type	*resume* type	Total	Graph density	Average egress edge per Activity	Number	w/ content provider	w/ input method	w/ network
WebMD	38	274	129	403	14.0%	10.0	92	18.7%	15.3%	70.3%
Chase	34	257	39	296	17.4%	8.71	50	4%	0.00%	44.0%
Amazon	19	209	190	399	55.3%	21.0	39	0.00%	7.69%	74.3%
NewEgg	55	242	253	495	8.2%	9.0	80	0.00%	8.75%	97.5%
GMail	7	10	10	20	20.4%	2.86	17	0.00%	5.71%	5.8%
H&R Block	20	58	39	97	12.1%	4.85	42	0.00%	2.3%	100%
Hotel.com	24	29	41	70	6.1%	2.92	35	0.00%	2.8%	100%

Table 2: Characteristics of Activity, Activity LandingStates and Activity transitions of selected apps (numbers are obtained manually as the ground truth). The CPU utilization time feature is omitted since it is always available.

Application name	Activity transition detection			Activity inference accuracy		
	Accuracy	FP	FN	Top 1 cand.	Top 2 cand.	Top 3 cand.
WebMD	99%	0.50%	1.0%	84.5%	91.4%	93.6%
Chase	99.5%	0.53%	0.63%	83.1%	91.8%	95.7%
Amazon	99.3%	4%	0.7%	47.6%	65.6%	74.1%
NewEgg	98.4%	0.1%	1.6%	85.9%	92.6%	96.3%
GMail	99.2%	0%	0.8%	92.0%	98.3%	99.3%
H&R Block	97.7%	2%	2.3%	91.9%	96.7%	98.1%
Hotel.com	96.5%	0.6%	3.5%	82.6%	92.7%	96.7%

Table 3: Activity transition detection and inference result for selected apps. All results are generated using Activity traces with more than 3000 transitions.

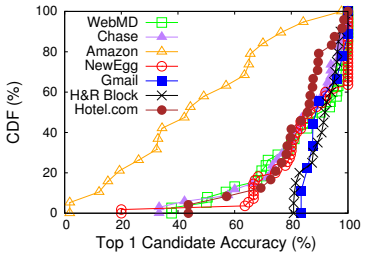


Figure 11: CDF of the average accuracy for top 1 candidates

5 Evaluation

In this section, we evaluate (1) the effectiveness of the automated Activity transition tool, (2) the performance of the Activity inference, and (3) the attack overhead.

Attack implementation. We implement the Activity inference attack with around 2300 lines of C++ code compiled with Android NDK packaged in an attack app.

Data collection. We use the automated tool in §4.3 to generate Activity transitions. We use random traversals to simulate user behavior, and deterministic traversals in controlled experiments for training and parameter selection, *e.g.,* the sampling rate. We run all experiments on Samsung Galaxy S3 devices with Android 4.2. We do not make use of any device-specific features and expect our findings to apply to other Android phones.

App selection and characteristics. In our experiments, we choose 7 Android apps, WebMD, GMail, Chase, H&R Block, Amazon, NewEgg, and Hotel.com, all of which are popular and also contain sensitive information. Table 2 characterizes these apps according to properties relevant to our attack techniques. NewEgg and GMail have the highest and the lowest number of Activities, and Amazon has the highest graph density. Chase app is the only one with no automatic soft keyboard pop-up during the transition among these apps. The Content Provider is only extensively used by WebMD. Except GMail, the percentage of the network feature is usually high.

5.1 Activity Transition Tool Evaluation

For Activity transition graph generation, the tool typically spends 10 minutes to 1 hour on a single Activity, depending on the UI complexity. For all apps except WebMD, the generated transition graphs are exactly the same as the ones we generate manually. The transition graph of WebMD misses 4 *create* transition edges and 3 *resume* transition edges, which is caused by dependent Activity issues described in §4.3. Our tool generates no fake edges for all selected apps.

5.2 Activity Inference Attack Evaluation

Evaluation methodology. We run the attack app in the background while the tool triggers Activity transitions. The triggered Activity traces are recorded as the ground truth. To simulate the real attack environment, the attack is launched with popular apps such as GMail and Facebook running in the background. For the Activity transition detection, we measure the accuracy, false positive (FP) and false negative (FN) rates. For the Activity inference, we consider the accuracy for the top N candidates — the inference is considered correct if the right Activity is ranked top N on the result candidate list.

5.2.1 Activity Transition Detection Results

Aggregated Activity transition detection results are shown in columns 2–4 in Table 3. For all selected apps, the detection accuracies are more than 96.5%, and the FP and FN rates are both less than 4%.

When changing the sampling period from 30 to 100 ms in our experiment, for all apps the increases of FP and FN rates are no more than 5%. This shows a small impact of the sampling rate on the detection; thus, a lower sampling rate can be used to reduce sampling overhead.

We also measure Activity transition detection delay, which is the time from the first *shared_vm* increase to the moment when the Activity transition is detected in

Fig. 10. For all apps, 80% of the delay is shorter than 1300 ms, which is fast enough for Activity tracking.

5.2.2 Activity Inference Results

The aggregated Activity transition inference result is shown in column 5–7 in Table 3. For all apps except Amazon, the average accuracies for the top 1 candidates are 82.6–92.0%, while the top 2 and top 3 candidates' accuracies exceed 91.4% and 93.6%. Amazon's accuracy remains poor, and can achieve 80% only when considering the top 5 candidates. In the next section, we will investigate more into the reason of these results.

Fig. 11 shows the CDF of the accuracy for top 1 candidates per Activity in the selected apps. Except Amazon, all apps have more than 70% of Activities with more than 80% accuracy. For WebMD, NewEgg, Chase and Hotel.com, around 20% Activities have less than 70% accuracy. For these Activities, they usually lack some signature features, or the features they have are too common to be distinct enough. However, such Activities usually do not have sensitive data due to a lack of important UI features such as text fields for typing, and thus are not relevant to the proposed attacks. For example, in Hotel.com, the two Activities with less than 70% accuracy are CountrySelectActivity for switching language and OpinionLabEmbeddedBrowserActivity for rating the app.

5.2.3 Breakdown Analysis and Discussion

To better understand the performance results, we break down the contributions of each signature feature and the transition model further. Table 4 shows the decrease of the average accuracy for top 1 candidates if leaving out certain features or the transition model. Without the CPU utilization time feature, the accuracy decreases by 36.2% on average, making it the most important contributor. Contributions from the network feature and the transition model are also high, which generally improves the accuracy by 12–30%. As low-entropy features, the Content Provider and the input method contribute under 5%. Thus, the CPU utilization time, the network event and the transition model are the three most important contributors to the final accuracy. Note that though the Content Provider and input method features have lower contributions, we find that the top 2 and top 3 candidates' accuracies benefit more from them. This is because they are more stable features, and greatly reduce the cases with extremely poor results due to the high variance in the CPU utilization time and the network features.

Thus, considering that the CPU utilization time is always available, apps with a high percentage of network features, or a sparse transition graph, or both, should have a high inference accuracy. In Table 2 and Table 3, this rule applies to all the selected apps except Amazon.

Application name	Δ Accuracy for top 1 candidates				
	no IM	no CP	no Net	no CPU	no HMM
WebMD	-3.8%	-2.6%	-19.1%	-25.7%	-16.6%
Chase	-0%	-2.0%	-12.8%	-71.5%	-28.7%
Amazon	-10.2%	-0%	-3.2%	-32.0%	-5.9%
NewEgg	-0.5%	-0%	-31.7%	-20.0%	-13.0%
GMail	-13.7%	-0%	-0.9%	-58.6%	-19.4%
H&RBlock	-0.7%	-0%	-30.7%	-27.9%	-16.5%
Hotel.com	-0.3%	-0%	-28.8%	-17.9%	-12.2%

Table 4: Breakdown of individual contributions to accuracy. IM, CP, Net, and CPU stand for input method, Content Provider, network event and CPU utilization time.

Amazon has a low accuracy mainly because it benefits little from either the transition model or the network event feature due to high transition graph density and infrequent network events. The reason for the high transition graph density is that in Amazon each Activity has a menu which provides options to transition to nearly all other Activities. The infrequent network events are due to its extensively usage of caching, presumably because much of its content is static and cacheable. However, we note that many network events are typically not cacheable, *e.g.,* authentication events and dynamic content (generated depending on the user input and/or the context). Compared to the other 6 apps, we find that these two properties for Amazon are not typical, not present in another shopping app NewEgg.

The Amazon app case indicates that our inference method may not work well if certain features are not sufficiently distinct, especially the major contributors such as the transition model and the network event feature. To better understand the general applicability of our inference technique, a more extensive measurement study about the Activity and Activity transition graph properties is needed, which we leave as future work.

5.2.4 Attack overhead

We use the Monsoon Power Monitor [28] to measure the attack energy overhead. Using an Activity trace of WebMD on the same device, with our attack in the background the power level increases by 2.2 to 6.0% when the sampling period changes from 100 to 30 ms.

6 Enabled Attack: Activity Hijacking

In this section, based on the UI state tracking, we design a new Android attack which breaches GUI integrity — Activity hijacking attack — based on a simple idea: stealthily inject into the foreground a phishing Activity at the right timing and steal sensitive information a user enters, *e.g.,* the password in login Activity.

Note that this is not the first attack popping up a phishing Activity to steal user input, but we argue that it is the first general one that can hijack any Activities during an app's lifetime. Previous study [29] pops up a fake login

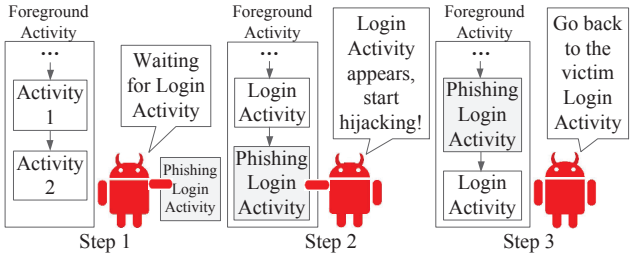


Figure 12: General steps of the Activity hijacking attack

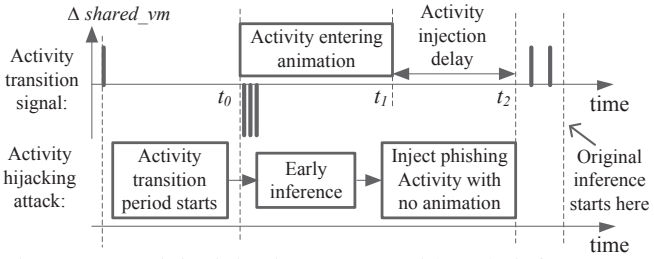


Figure 13: Activity injection process with early inference

Activity every time the attack app detects the launching of the target app, tricking users into entering login credentials. However, this can easily cause user suspicion due to the following: (1) most apps nowadays do not require login right after the app starts, even for banking apps like Chase; (2) the attack requires suspicious permissions such as BOOT_COMPLETED to be notified of system boot, based on the assumption that login is expected after the system reboot. With the Activity inference attack, we no longer suffer from these limitations.

6.1 Activity Hijacking Attack Overview

Fig. 12 shows the general steps of Activity hijacking attack. In step 1, the background attack app uses Activity inference to monitor the foreground Activity, waiting for the attack target, for example, LoginActivity in Fig. 12.

In step 2, once the Activity inference reports that the target victim Activity, *e.g.,* LoginActivity, is about to enter the foreground, the attack app simultaneously injects a pre-prepared phishing LoginActivity into the foreground. Note that the challenge here is that this introduces a race condition where the injected phishing Activity might enter the foreground too early or too late, causing visual disruption (*e.g.,* broken animation). With carefully designed timing, we prepare the injection at the perfect time without any human-observable glitches during the transition (see video demos [6]). Thus, the user will not notice any differences, and continue entering the password. At this point, the information is stolen and the attack succeeds.

In step 3, the attack app ends the attack as unsuspiciously as possible. Since the attack app now becomes the foreground app, it needs to somehow transition back to the original app without raising much suspicion.

6.2 Attack Design Details

Activity injection. To understand how it is possible to inject an Activity from one app into the foreground and preempt the existing one, we have to understand the design principle of smartphone UI. If we think about apps such as the alarm and reminder apps, they indeed require the ability to pop up a window and preempt any foreground Activities. In Android, such functionality is supported in two ways without requiring any permissions: (1) starting an Activity with a restricted launching mode "SingleInstance" [30]; (2) starting an Activity from an Android broadcast receiver [31]. In our design, since the timing of the injection is critical, we choose the former as it can be launched 30 ms faster.

UI phishing. To ensure that the phishing Activity's UI appears the same as the victim Activity, we disassemble the victim app's apk using *apktool* [32] and copy all related UI resources to the attack app. However, sometimes the Activity UI may have dynamically loaded areas which are not determined by the UI resources, *e.g.,* the account verification image in banking apps. To solve that, the attacker can make those areas transparent, given that Android supports partially transparent Activity [33].

Activity transition animation modifying. Since our injection introduces an additional Activity transition which is racing with the original transition, the animation of the additional transition would clearly disrupt the flow. Fortunately, this problem can be solved by disabling the transition animation (allowed by Android) by modifying an Activity configuration of the attack app without needing any permissions. This helps the injection become totally seamless, and as will be discussed in §9, enforcing this animation may be a feasible mitigation of our attack.

Injection timing constraint. For the attack to succeed, the Activity injection needs to happen before any user interaction begins, otherwise the UI change triggered by it may be disrupted by the injected Activity. Since the injection with the current inference technique takes quite long (the injected Activity will show up after around 1300 ms from the first detected *shared_vm* increase as measured in §5), any user interaction during this period would cause disruptions. To reduce the delay, we adapt the inference to start much earlier. As shown in Fig. 13, we now start the inference as soon as the *shared_vm* decrease is observed (roughly corresponding to the Activity entering animation start time). In contrast, our original inference starts after the last *shared_vm* increase.

Note that this would limit the feature collection up to the point of the *shared_vm* decrease, thus impacting the inference accuracy. Fortunately, as indicated in Fig. 10, such change does allow the network event feature, the

majority of the CPU utilization time features, and the transition model to be included in the inference, which are the three most important contributors to the final accuracy as discussed in §5.2.3. Based on our evaluation, this would reduce the delay to only around 500 ms.

Unsuspicious attack ending. As described in §6.1, in step 3 we try to transition from the attack app back to the victim unsuspiciously. Since the phishing Activity now has the information entered on the screen, it is too abrupt to directly close it and jump back to the victim. In our design, we leverage "benign" abnormal events to hide the attack behavior, *e.g.,* the attack app can show "server error" after the user clicks on the login button, and then quickly destroy itself and fall back to the victim.

Deal with cached user data. It is common that some sensitive data may be cached, thus won't be entered at all, *e.g.,* the user name in login Activity. Without waiting for them to expire, it is difficult to capture any fresh input.

Note that we can simply inject the phishing Activity with all fields left blank. The challenge is to not alert the user with any other observable suspicious behavior. Specifically, depending on the implementation, we find that the cached user data sometimes show up immediately in the very first frame of the Activity entering animation (t_0 in Fig. 13). Thus, our later injection would clear the cached fields, which causes visual disruption.

Our solution is to pop up a tailored cache expiration message (replicating the one from the app), and then clear such cached data, prompting users to re-enter them.

6.3 Attack Implementation and Evaluation

Implementation. We implement Activity hijacking attack against 4 Activities: H&R Block's LoginActivity and RefundInfoActivity for stealing the login credentials and SSN, and NewEgg's ShippingAddressAddActivity and PaymentOptionsModifyActivity for stealing the shipping/billing address and credit card information. The latter two Activities do not appear frequently in the check-out process since the corresponding information may be cached. Thus, to force the user to re-enter them, our attack injects these two Activities into the check-out process. The user would simply think that the cached information has expired. In this case the fake cache expiration messages are not needed, since the attack can fall back to the check-out process naturally after entering that information. Attack demos can be found in [6].

Evaluation. The most important metric for our attack is the Activity injection delay, which is the time from t_1 to t_2 in Fig. 13. In Android, it is hard to know precisely the animation ending time t_1, so the delay is measured from t_0 to t_2 as an upper bound. In the evaluation the Activity injection is performed 50 times for the LoginActivity of H&R Block app, and the average injection delay is 488 ms. Most of the delay time is spent in `onCreate()` (242 ms) and `performTraverse()` (153 ms). From our experience, the injection is fast enough to complete before any user interaction starts.

7 Enabled Attack: Camera Peeking

In this section, we show another new attack enabled by the Activity inference: camera peeking attack.

7.1 Camera Peeking Attack Overview

Due to privacy concerns, many apps store photo images shot by the camera only in memory and never make them publicly accessible, for example by writing them to external storage. This applies to many apps such as banking apps (*e.g.,* Chase), shopping apps (*e.g.,* Amazon and Best Buy), and search apps (*e.g.,* Google Goggles). Such photo images contain highly-sensitive information such as the user's life events, shopping interests, home address and signature (on the check). Surprisingly, we show that with Activity tracking such sensitive and well-protected camera photo images can be successfully stolen by a background attack app. Different from PlaceRaider [34], our attack targets at the camera photo shot by the user, instead of random ones of the environment.

Our attack follows a simple idea: when an Activity is using the camera, the attack app quickly takes a separate image while the camera is still in the same position. In the following, we detail our design and implementation.

7.2 Attack Design Details

Background on Android camera resource management. With the camera permission, an Android app can obtain and release the camera by calling `open()` and `release()`. Once the camera is obtained, an app can then take pictures by calling `takePicture()`. There are two important properties: (1) exclusive usage. The camera can be used by only one app at any point in time; (2) slow initialization. Camera initialization needs to work with hardware, so `open()` typically takes 500–1000 ms (measured on Samsung Galaxy S3 devices).

Obtain camera images in the background. In the Android documentation, taking pictures in the background is explicitly disallowed due to privacy concerns [35]. Though PlaceRaider [34] succeeded in doing so, we find that their technique is restricted to certain devices running old Android systems which do not follow the documentation correctly, *e.g.,* Droid 3 with Android 2.3.

Interestingly, we find *camera preview frames* to be the perfect alternative interface for obtaining camera images without explicitly calling `takePicture()`. When using the camera, the preview on the screen shows a live video stream from the camera. Using `PreviewCallback()`, the video stream frames are returned one by one, which are actually the camera images we want. SurfaceTexture is used to capture this

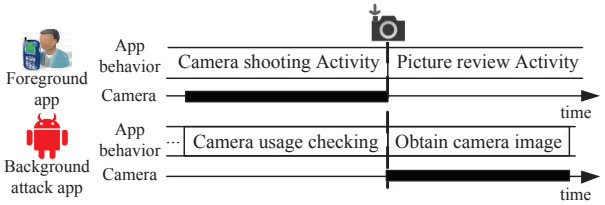


Figure 14: Camera peeking attack process when the foreground Activity is using the camera

Camera peeking attack type	Success rate	DoS rate	# of camera possession per round
Blind attack (3s idle time)	81%	19%	30.5
Blind attack (4s idle time)	83%	14%	20.9
Blind attack (5s idle time)	79%	8%	18.9
UI state based attack	99%	0%	1.4

Table 5: User study evaluation result for the camera peeking attack

image stream, and we find that it can be created with a nonexistent OpenGL texture object name, thus preventing any visible preview on the screen. We suspect that the less restrictive interface is managed by OpenGL library which bypasses the Android framework and its associated security checks. Compared to PlaceRaider [34], this technique not only has no requirement of the sensitive MODIFY_AUDIO_SETTINGS permission to avoid shutter sound, but also has much faster "shutter speed" of 24 frames per second. Note that even if this interface is blocked, our attack can still use techniques in §6 to inject an Activity to the foreground to get the preview frames.

Obtain photo images shot by the user. Fig. 14 shows how our attack gets the photo image the user shoots in the victim app. The photo taking functionality usually involves a camera shooting Activity and a picture review Activity. Once the user clicks on the shutter button in the former, the latter pops up with the picture just taken. Due to the exclusive usage property, when the foreground Activity is using the camera the attack app cannot get the camera. Thus, once knowing that the camera is in use, the attack app keeps calling `open()` to request the camera until it succeeds right after the user presses the shutter button and the camera gets released during the Activity transition. Since the delay to get a camera preview frame is only the initialization time (500–1000 ms), the camera is very likely still pointing at the same object, thus obtaining a similar image.

Capture the camera usage time. To trigger the attack process in Fig. 14, the attack app needs to know when the camera is in use in the foreground. However, due to the slow initialization, a naive solution which periodically calls `open()` to check the camera usage will possess the camera for 500–1000 ms for each checking action when the camera is not in use. During that time, if the foreground app tries to use the camera, a denial of service (DoS) will take place. With 12 popular apps, we find that when failing to get the camera, most apps pop up a "camera error" or "camera is used by another app" message and some even crash immediately. These errors may indicate that an attack is ongoing and alert the user. Besides, the frequent camera resource possessing behavior is easily found suspicious with increasing concerns about smartphone camera usage [34].

To solve the problem, our attack uses Activity inference to capture the camera usage time by directly waiting for the camera shooting Activity. To increase the inference accuracy for Activities using the camera, we add camera usage as a binary feature (true or false on the camera usage status) and it is only tested when the landing Activity is very likely to be the camera shooting Activity based on other features to prevent DoS and overly frequent camera possessions.

7.3 Attack Evaluation

Implementation. We implement the camera peeking attack against the check deposit functionality in Chase app, which allows users to deposit personal checks by taking a picture of the check. Besides the network permission, the attack app also needs the the camera permission to access camera preview frames. On the check photo, the attacker can steal much highly-sensitive personal information, including the user name, home address, bank routing/account number, and even the user's signature. A video demo is available at [6].

Evaluation methodology. We compare our UI state based camera peeking attack against the blind attack, which periodically calls `open()` to check the foreground camera usage as described in §7.2. We add parameter *idle time* for the blind attack as the camera usage checking period. The longer the idle time is, the lower the DoS possibility and the camera possession frequency are. However, the idle time cannot be so long that the attack misses the camera shooting events. Thus, the blind attack faces a trade-off between the DoS risk, the camera possession frequency, and the attack success rate.

User study. We evaluate our attack with a user study of 10 volunteers. In the study we use 4 Samsung Galaxy S3 phones with Android 4.1. Three of them use the blind attacks with idle time being 3, 4 and 5 seconds respectively, and the last one uses the UI state based attack. Each user performs 10 rounds, and in each round, the users are asked to first interact with the app as usual, and then go to the check deposit Activity and take a picture of a check provided by us. We emphasize that they are expected to perform as if it is their own check. The IRB for this study has been approved and we took steps to ensure that no sensitive user data were exposed, *e.g.,* by using a fake bank account and personal checks.

Performance metrics. For evaluation we measure: (1)

DoS rate, the ratio that when the user wants to use the camera but fails; (2) number of camera possessions, the number of events that the camera is possessed by the attack app; (3) success rate, the ratio that the attacker gets the check image after the user shoots the check.

Result. Table 5 shows the user study evaluation results. With the camera usage feature, the UI state based attack can achieve 99% success rate, and the only failure case is due to a failure in detecting the Activity transition. For the blind attack, the success rate is less than 83%, and when the idle time increases, the success rate increases then decreases. The increase is due to lower DoS probability, and the decrease is because the users usually finish shooting in around 4 seconds (found in our user study), so when the idle time increases to 5 seconds, the blind attack misses some camera shooting events.

UI state based attack causes no DoS during the user study. For the blind attack, the DoS rate is around 8–19%, and decreases when the idle time increases. Considering that a single DoS case may likely cause "sudden death" for the attack app, this risk is high, especially compared to the UI state based attack.

The camera possession number for the UI state based attack is also a magnitude lower. Every round, except the necessary one for camera shooting, the UI state based attack only needs 0.4 excessive camera possessions, which is mainly caused by inaccurate inference. For the blind attack, to ensure a high success rate, the camera possession number is proportional to time, making it hard to avoid suspicious frequent camera possessions.

Fig. 15 includes an average quality check image "stolen" from a real user, showing that the image is clear enough to read all the private information.

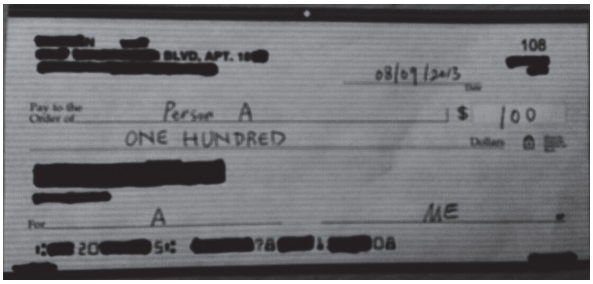


Figure 15: An example check image "stolen" using the camera peeking attack.

8 Other Attack Scenarios

Enhance existing attacks. Generally, a class of existing attacks that are launched only at specific timings benefits from UI state information. Since many attacks need to be launched at a specific timing, with the UI state information, both stealthiness and effectiveness can be achieved. For example, for the phishing attack using TCP connection hijacking [9, 10], the attack app can precisely target at connections established in Activities with web pages instead of unrelated ones, *e.g.,* database updating, keepalive messages, *etc.* The attack thus becomes more efficient and less suspicious by avoiding frequently sending large amounts of attack traffic [9]. Similar enhancement can also be applied to keystroke inference attacks [7, 8] and screenshot taking attack [5] where only keystrokes entered in login Activities may be of interest.

User behavior monitoring and analysis. UI states themselves are user behavior related internal states of an app. As shown in Fig. 2, due to the limited screen size on the smartphone, full-screen window-level UI state information breaks user-app interaction to very fine-grained states. Thus, by tracking UI states, a background app can monitor and analyze user behavior, *e.g.,* in a health app the user is more often looking for drugs or physicians.

In addition, with Activity tracking, the attacker can even infer which choice is made inside an Activity (*e.g.,* which medical condition a user is interested in). This is achieved using the size of the request packet obtained by the technique described in §4.1. For example, for QAListActivity of H&R Block app, we can infer which tax question a user is interested in based on the length of the question that is embedded in the query packet. In this question list, we find that 10 out of 11 question queries are distinguishable (with different lengths).

A similar technique was proposed recently [12], but built upon a network event based state machine, with two limitations: (1) packet size itself can be highly variable (ads connections may co-occur) and different Activities may generate similar packet size traces, *e.g.,* login Activities and user account Activities both have the authentication connection thus may have similar packet size trace. UI state knowledge would limit the space of possible connections significantly as we infer the Activity based on a more comprehensive set of features; (2) not all user choices in Android are reflected in network packets — database/Content Provider can also be used to fetch content. With our UI state machine, we can further extend the attack to the Content Provider based user choice inference. For example, in WebMD, DrugSearchMainActivity has a list of letter A to Z. Once one letter is clicked, Content Provider is queried to fetch a list of drug names starting from that letter. With the Content Provider query event and content size inference technique (described in §4.1), we characterized all of the choices and found fairly good entropy: the responding content sizes have 16 different values for the 26 letters, corresponding to 4 bits out of 4.7 bits of information for the user choice.

9 Defense Discussion

In this section, we suggest more secure system designs to defend against the attacks found in this paper.

Proc file system access control. In our attack, *shared_vm* and features of Activity signature such as CPU and network all rely on freely accessible files in proc file system. However, as pointed out by Zhou *et al.* [12], simply removing them from the list of public resources may not be a good option due to the large amount of existing apps depending on them. To better solve the problem, Zhou *et al.* [12] proposed a mitigation strategy which reduces the attack effectiveness by rounding up or down the actual value. This can work for the network side channel, but may not be effective for *shared_vm* and *shared_pm*, which are already rounded to pages (4KB) but still generate a high transition detection accuracy. This is mainly because the window buffer size is large enough to be distinct and the side channel is pretty clean, as discussed in §3.3. Thus, Android system may need to reconsider its access control strategy for these public accessible files to better balance functionality and security. In fact, Android has already restricted access to certain proc files that are publicly accessible in the standard Linux, *e.g., /proc/pid/smaps*. However, our work indicates that it is still far from being secure.

Window manager design. As described in §3.2, the existence of the shared-memory side channel is due to the requirement of the window buffer sharing in the client-drawn buffer design. Thus, a fundamental way of defending against the UI state inference attack in this paper is to use the server-drawn buffer design in GUI systems, though this means that any applications that are exposed to the details of the client-drawn buffer design need to be updated, which may introduce other side effects.

Window buffer reuse. The Activity transition signal consists of *shared_vm* increases and decreases, corresponding to window buffer allocations and deallocations. To eliminate such signal, the system can avoid them by pre-allocating two copies of the buffers and reuse them for all transitions in an app. Note that this is at the cost of much more memory usage for each app, as each buffer is several megabytes in size. However, with increasingly larger memory size in future mobile devices [36], we think this overhead may be acceptable.

In this paper, the most serious security breaches are caused by follow-up attacks based on UI state inference. Thus, we provide suggestions as follows that can mitigate the attacks even if the UI state information is leaked.

Enforce UI state transition animation. Animation is an important indicator for informing users about app state changes. In the Activity hijacking attack in §6, the seamless Activity injection is possible because this indicator can be turned off in Android. With UI state tracking, the attacker can leverage this to replace the foreground UI state with a phishing one without any visible indications. Thus, one defense on GUI system design side is to always keep this indicator on by enforcing animation in all UI state transitions. This helps reduce the attack stealthiness though it cannot fully eliminate the attack.

Limit background application functionality. In GUI systems, background applications do not directly interact with users, so they should not perform privacy-sensitive actions freely. In §7, a background attacker can still get camera images, indicating that Android did not sufficiently restrict the background app functionality. With UI state tracking, an attacker can leverage precise timing to circumvent app data isolation. Thus, more restrictions should be imposed to limit background applications' access to sensitive resources like camera, GPS, sensor, *etc.*

To summarize, we propose solutions that eliminate dependencies of the attack such as the proc file side channel, which may prevent the attack. However, more investigation is required to understand their effectiveness and most of them do require significant changes that have impact on either backward-compatibility or functionality.

10 Related Work

Android malware. The Android OS, like any systems, contains security vulnerabilities and is vulnerable to malware [37–39]. For instance, the IPC mechanisms leave Android vulnerable to confused deputy attacks [38, 39]. Malware can collect privacy-sensitive information by requesting certain permissions [37, 40]. To combat these flaws, a number of defenses have been proposed [38, 41], such as tracking the origin of inter-process calls to prevent unauthorized apps from indirectly accessing privileged information. Our attack requires neither specific vulnerabilities nor privacy-sensitive permissions, so known defense mechanisms will not protect against it.

Side-channel attacks. Much work has been done on studying side channels. **Proc file systems** have been long abused for side-channel attacks. Zhang *et al.* [24] found that the ESP/EIP value can be used to infer keystrokes. Qian *et al.* [10] used "sequence-number-dependent" packet counter side channels to infer TCP sequence number. In memento [11], the memory footprints were found to correlate with the web pages a user visits. Zhou *et al.* [12] found three Android/Linux public resources to leak private information. These attacks are mostly app-dependent, while in this paper the UI state inference applies generally to all Android apps, leading to not only a larger attack coverage but also many more serious attacks. **Timing** is another popular form of side channels. Studies have shown that timing can be used to infer keystrokes as well as user information revealed by web applications [23, 42–44]. **Sensors** are more recent, popular side-channel sources. The sound made by the keyboard [45], electromagnetic waves [46], and special software [47] can be used to infer keystrokes. More recently, a large number of sensor-based side channels have been discovered on Android, including the micro-

phone [18], accelerometer [7, 8] and camera [34]. Our attack does not rely on sensors which may require suspicious permissions. Instead, we leverage only data from the proc file system, which is readily available with no permission requirement.

Root causes of side-channel attacks. All side-channel attacks exist because of certain behavior in the software/hardware stack that can be distinguished through some forms of observable channels by attackers. For example, the inter-keystroke timing attack exploits the application and OS behavior that handles user input. SSH programs will send whatever keys the user types immediately to the network, so the timing is observable through a network packet trace [23]. For VIM-like programs, certain program routines are triggered whenever a new key is captured, so the timing can be captured through snapshots of the program's ESP/EIP values [24]. The TCP sequence number inference attack [10] exploits the TCP stack of the OS that exposes internal states through observable packet counters. In our attack, we exploit a new side channel caused by popular GUI framework behavior, in particular how user interaction and window events are designed and implemented.

11 Conclusion

In this paper, we formulate the UI state inference attack designed at exposing the running UI state of an application. This attack is enabled by a newly-discovered shared-memory side channel, which exists in nearly all popular GUI systems. We design and implement the Android version of this attack, and show that it has a high inference accuracy by evaluating it on popular apps. We then show that UI state tracking can be used as a powerful attack building block to enable new Android attacks, including Activity hijacking and camera peeking. We also discuss ways of eliminating the side channel, and suggest more secure system designs.

Acknowledgments

We would like to thank Sanae Rosen, Denis Foo Kune, Zhengqin Luo, Yu Stephanie Sun, Earlence Fernandes, Mark S. Gordon, the anonymous reviewers, and our shepherd, Jaeyeon Jung, for providing valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-131830, CNS-1059372, CNS-1039657, CNS-1345226, and CNS-0964545, as well as by the ONR grant N00014-14-1-0440.

References

[1] N. Feske and C. Helmuth, "A Nitpickers guide to a minimal-complexity secure GUI," in *ACSAC*, 2005.

[2] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, "Design of the EROS trusted window system," in *USENIX Security Symposium*, 2004.

[3] T. Fischer, A.-R. Sadeghi, and M. Winandy, "A pattern for secure graphical user interface systems," in *20th International Workshop on Database and Expert Systems Application*. IEEE, 2009.

[4] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang, "A Systematic Approach to Uncover Security Flaws in GUI Logic," in *IEEE Symposium on Security and Privacy*, 2007.

[5] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to Milk Your Android Screen for Secrets," in *NDSS*, 2014.

[6] "Video Demos for this Paper," https://sites.google.com/site/uistateinferenceattack/demos.

[7] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *WiSec*, 2012.

[8] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *Mobisys*, 2012.

[9] Z. Qian and Z. M. Mao, "Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security," in *IEEE Symposium on Security and Privacy*, 2012.

[10] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative tcp sequence number inference attack: how to crack sequence number under a second," in *CCS*, 2012.

[11] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *IEEE Symposium on Security and Privacy*, 2012.

[12] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources," in *CCS*, 2013.

[13] "Wayland," http://wayland.freedesktop.org/.

[14] "Ubuntu Move to Wayland," http://www.markshuttleworth.com/archives/551.

[15] "Mir," https://wiki.ubuntu.com/Mir.

[16] "Back Stack," http://developer.android.com/guide/components/tasks-and-back-stack.html.

[17] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *OOPSLA*, 2013.

[18] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *NDSS*, 2011.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *CCS*, 2011.

[20] "VMMap," http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx.

[21] "project-butter," http://www.androidpolice.com/2012/07/12/getting-to-know-android-4-1-part-3-project-butter-how-it-works-and-what-it-added/.

[22] "Android DDMS," http://developer.android.com/tools/debugging/ddms.html.

[23] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.

[24] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.

[25] "Whois IP Address Database," http://whois.net/.

[26] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[27] "Android Activity Testing," http://developer.android.com/tools/testing/activity_testing.html.

[28] "Monsoon Power Monitor," http://www.msoon.com/LabEquipment/PowerMonitor/.

[29] "Focus Stealing Vulnerability," http://blog.spiderlabs.com/2011/08/twsl2011-008-focus-stealing-vulnerability-in-android.html.

[30] "Android Launching Mode," http://developer.android.com/guide/topics/manifest/activity-element.html#lmode.

[31] "Android Broadcast Receiver," http://developer.android.com/reference/android/content/BroadcastReceiver.html.

[32] "Android Apktool," http://code.google.com/p/android-apktool/.

[33] "Transparent Activity Theme," http://developer.android.com/guide/topics/ui/themes.html#ApplyATheme.

[34] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, "PlaceRaider: Virtual Theft in Physical Spaces with Smartphones," in *NDSS*, 2013.

[35] "Android Camera," http://developer.android.com/reference/android/hardware/Camera.html.

[36] "Samsung Wants to Cram 4GB of RAM into Your Next Phone," http://www.pcworld.com/article/2083320/samsung-lays-groundwork-for-smartphones-with-more-ram.html.

[37] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, 2012.

[38] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-delegation: Attacks and Defenses," in *USENIX Security Symposium*, 2011.

[39] Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," in *NDSS*, 2013.

[40] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information Flow Tracking System for Real-Time Privacy Monitoring on Smartphones," in *OSDI*, 2010.

[41] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *USENIX Security Symposium*, 2011.

[42] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," in *IEEE Symposium on Security and Privacy*, 2010.

[43] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, 2012.

[44] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *WWW*, 2007.

[45] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," in *CCS*, 2005.

[46] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards," in *USENIX security symposium*, 2009.

[47] K. Killourhy and R. Maxion, "Comparing Anomaly-Detection Algorithms for Keystroke Dynamic," in *DSN*, 2009.

Gyrophone: Recognizing Speech From Gyroscope Signals

Yan Michalevsky Dan Boneh
Computer Science Department
Stanford University

Gabi Nakibly
National Research & Simulation Center
Rafael Ltd.

Abstract

We show that the MEMS gyroscopes found on modern smart phones are sufficiently sensitive to measure acoustic signals in the vicinity of the phone. The resulting signals contain only very low-frequency information (<200Hz). Nevertheless we show, using signal processing and machine learning, that this information is sufficient to identify speaker information and even parse speech. Since iOS and Android require no special permissions to access the gyro, our results show that apps and active web content that cannot access the microphone can nevertheless eavesdrop on speech in the vicinity of the phone.

1 Introduction

Modern smartphones and mobile devices have many sensors that enable rich user experience. Being generally put to good use, they can sometimes unintentionally expose information the user does not want to share. While the privacy risks associated with some sensors like a microphone (eavesdropping), camera or GPS (tracking) are obvious and well understood, some of the risks remained under the radar for users and application developers. In particular, access to motion sensors such as gyroscope and accelerometer is unmitigated by mobile operating systems. Namely, every application installed on a phone and every web page browsed over it can measure and record these sensors without the user being aware of it.

Recently, a few research works pointed out unintended information leaks using motion sensors. In Ref. [34] the authors suggest a method for user identification from gait patterns obtained from a mobile device's accelerometers. The feasibility of keystroke inference from nearby keyboards using accelerometers has been shown in [35]. In [21], the authors demonstrate the possibility of keystroke inference on a mobile device using accelerometers and mention the potential of using gyroscope measurements as well, while another study [19] points to the benefits of exploiting the gyroscope.

All of the above work focused on exploitation of motion events obtained from the sensors, utilizing the expected kinetic response of accelerometers and gyroscopes. In this paper we reveal a new way to extract information from gyroscope measurements. We show that gyroscopes are sufficiently sensitive to measure acoustic vibrations. This leads to the possibility of recovering speech from gyroscope readings, namely using the gyroscope as a crude microphone. We show that the sampling rate of the gyroscope is up to 200 Hz which covers some of the audible range. This raises the possibility of eavesdropping on speech in the vicinity of a phone without access to the real microphone.

As the sampling rate of the gyroscope is limited, one cannot fully reconstruct a comprehensible speech from measurements of a single gyroscope. Therefore, we resort to automatic speech recognition. We extract features from the gyroscope measurements using various signal processing methods and train machine learning algorithms for recognition. We achieve about 50% success rate for speaker identification from a set of 10 speakers. We also show that while limiting ourselves to a small vocabulary consisting solely of digit pronunciations ("one", "two", "three", ...) and achieve speech recognition success rate of 65% for the speaker dependent case and up to 26% recognition rate for the speaker independent case. This capability allows an attacker to substantially leak information about numbers spoken over or next to a phone (i.e. credit card numbers, social security numbers and the like).

We also consider the setting of a conference room where two or more people are carrying smartphones or tablets. This setting allows an attacker to gain simultaneous measurements of speech from several gyroscopes. We show that by combining the signals from two or more phones we can increase the effective sampling rate of the acoustic signal while achieving better speech recognition rates. In our experiments we achieved 77% successful recognition rate in the speaker dependent case based on the digits vocabulary.

The paper structure is as follows: in Section 2 we provide a brief description of how a MEMS gyroscope works and present initial investigation of its properties as a microphone. In Section 3 we discuss speech analysis and describe our algorithms for speaker and speech recognition. In Section 4 we suggest a method for audio signal recovery using samples from multiple devices. In Section 5 we discuss more directions for exploitation of gyroscopes' acoustic sensitivity. Finally, in Section 6 we discuss mitigation measures of this unexpected threat. In

particular, we argue that restricting the sampling rate is an effective and backwards compatible solution.

2 Gyroscope as a microphone

In this section we explain how MEMS gyroscopes operate and present an initial investigation of their susceptibility to acoustic signals.

2.1 How does a MEMS gyroscope work?

Standard-size (non-MEMS) gyroscopes are usually composed of a spinning wheel on an axle that is free to assume any orientation. Based on the principles of angular momentum the wheel resists to changes in orientation, thereby allowing to measure those changes. Nonetheless, all MEMS gyros take advantage of a different physical phenomenon – the Coriolis force. It is a fictitious force (d'Alembert force) that appears to act on an object while viewing it from a rotating reference frame (much like the centrifugal force). The Coriolis force acts in a direction perpendicular to the rotation axis of the reference frame and to the velocity of the viewed object. The Coriolis force is calculated by $F = 2m\vec{v} \times \vec{\omega}$ where m and v denote the object's mass and velocity, respectively, and ω denotes the angular rate of the reference frame.

Generally speaking, MEMS gyros measure their angular rate (ω) by sensing the magnitude of the Coriolis force acting on a moving proof mass within the gyro. Usually the moving proof mass constantly vibrates within the gyro. Its vibration frequency is also called the resonance frequency of the gyro. The Coriolis force is sensed by measuring its resulting vibration, which is orthogonal to the primary vibration movement. Some gyroscope designs use a single mass to measure the angular rate of different axes, while others use multiple masses. Such a general design is commonly called *vibrating structure gyroscope*.

There are two primary vendors of MEMS gyroscopes for mobile devices: STMicroelectronics [15] and InvenSense [7]. According to a recent survey [18] STMicroelectronics dominates with 80% market share. Teardown analyses show that this vendor's gyros can be found in Apple's iPhones and iPads [17, 8] and also in the latest generations of Samsung's Galaxy-line phones [5, 6]. The second vendor, InvenSense, has the remaining 20% market share [18]. InvenSense gyros can be found in Google's latest generations of Nexus-line phones and tablets [14, 13] as well as in Galaxy-line tablets [4, 3]. These two vendors' gyroscopes have different mechanical designs, but are both noticeably influenced by acoustic noise.

2.1.1 STMicroelectronics

The design of STMicroelectronics 3-axis gyros is based on a single driving (vibrating) mass (shown in Figure 1). The driving mass consists of 4 parts M_1, M_2, M_3 and M_4 (Figure 1(b)). They move inward and outward simultaneously at a certain frequency[1] in the horizontal plane. As shown in Figure 1(b), when an angular rate is applied on the Z-axis, due to the Coriolis effect, M_2 and M_4 will move in the same horizontal plane in opposite directions as shown by the red and yellow arrows. When an angular rate is applied on the X-axis, then M_1 and M_3 will move in opposite directions up and down out of the plane due to the Coriolis effect. When an angular rate is applied to the Y-axis, then M_2 and M_4 will move in opposite directions up and down out of the plane. The movement of the driving mass causes a capacitance change relative to stationary plates surrounding it. This change is sensed and translated into the measurement signal.

2.1.2 InvenSense

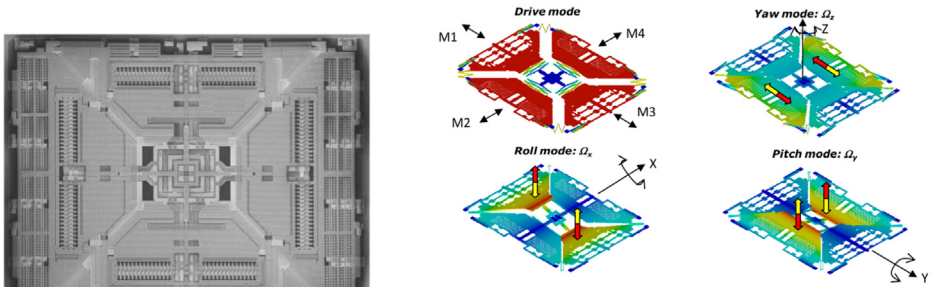
InvenSense's gyro design is based on the three separate driving (vibrating) masses[2]; each senses angular rate at a different axis (shown in Figure 2(a)). Each mass is a coupled dual-mass that move in opposite directions. The masses that sense the X and Y axes are driven out-of-plane (see Figure 2(b)), while the Z-axis mass is driven in-plane. As in the STMicroelectronics design the movement due to the Coriolis force is measures by capacitance changes.

2.2 Acoustic Effects

It is a well known fact in the MEMS community that MEMS gyros are susceptible to acoustic noise which degrades their accuracy [22, 24, 25]. An acoustic signal affects the gyroscope measurement by making the driving mass vibrate in the sensing axis (the axis which senses the Coriolis force). The acoustic signal can be transferred to the driving mass in one of two ways. First, it may induce mechanical vibrations to the gyros package. Additionally, the acoustic signal can travel through the gyroscope packaging and directly affect the driving mass in case it is suspended in air. The acoustic noise has the most substantial effect when it is near the resonance frequency of the vibrating mass. Such effects in some cases can render the gyro's measurements useless or even saturated. Therefore to reduce the noise effects vendors manufacture gyros with a high resonance frequency (above

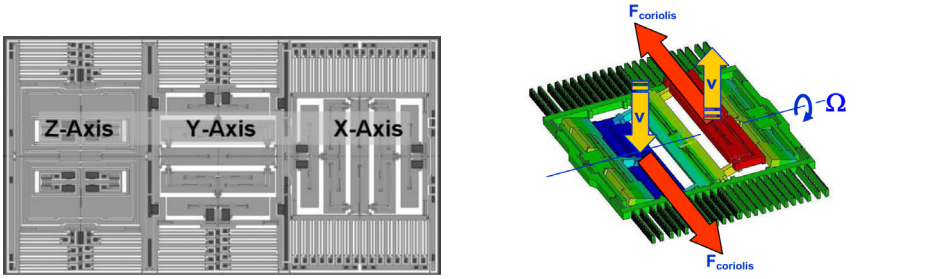
[1] It is indicated in [1] that STMicroelectronics uses a driving frequency of over 20 KHz.

[2] According to [43] the driving frequency of the masses is between 25 KHz and 30 KHz.



(a) MEMS structure (b) Driving mass movement depending on the angular rate

Figure 1: STMicroelectronics 3-axis gyro design (Taken from [16]. Figure copyright of STMicroelectronics. Used with permission.)



(a) MEMS structure (b) Driving mass movement depending on the angular rate

Figure 2: InvenSense 3-axis gyro design (Taken from [43]. Figure copyright of InvenSense. Used with permission.)

20 KHz) where acoustic signals are minimal. Nonetheless, in our experiments we found that acoustic signals at frequencies much lower than the resonance frequency still have a measurable effect on a gyro's measurements, allowing one to reconstruct the acoustic signal.

2.3 Characteristics of a gyro as a microphone

Due to the gyro's acoustic susceptibility one can treat gyroscope readings as if they were audio samples coming from a microphone. Note that the frequency of an audible signal is higher than 20 Hz, while in common cases the frequency of change of mobile device's angular velocity is lower than 20 cycles per second. Therefore, one can high-pass-filter the gyroscope readings in order to retain only the effects of an audio signal even if the mobile device is moving about. Nonetheless, it should be noted that this filtering may result in some loss of acoustic information since some aliased frequencies may be filtered out (see Section 2.3.2). In the following we explore the gyroscope characteristics from a standpoint of an acoustic sensor, i.e. a microphone. In this section we exemplify these characteristics by experimenting with Galaxy S III which has an STMicroelectronics gyro [6].

2.3.1 Sampling

Sampling resolution is measured by the number of bits per sample. More bits allow us to sample the signal more accurately at any given time. All the latest generations of gyroscopes have a sample resolution of 16 bits [9, 12]. This is comparable to a microphone's sampling resolution used in most audio applications.

Sampling frequency is the rate at which a signal is sampled. According to the Nyquist sampling theorem a sampling frequency f enables us to reconstruct signals at frequencies of up to $f/2$. Hence, a higher sampling frequency allows us to more accurately reconstruct the audio signal. In most mobile devices and operating systems an application is able to sample the output of a microphone at up to 44.1 KHz. A telephone system (POTS) samples an audio signal at 8000 Hz. However, STMicroelectronics' gyroscope hardware supports sampling frequencies of up to 800 Hz [9], while InvenSense gyros' hardware support sampling frequency up to 8000 Hz [12]. Moreover, all mobile operating systems bound the sampling frequency even further – up to 200 Hz – to limit power consumption. On top of that, it appears that some browser toolkits limit the sampling frequency even further. Table 1 summarizes the results of our experi-

		Sampling Freq. [Hz]
Android 4.4	application	200
	Chrome	25
	Firefox	200
	Opera	20
iOS 7	application	100 [2]
	Safari	20
	Chrome	20

Table 1: Maximum sampling frequencies on different platforms

ments measuring the maximum sampling frequencies allowed in the latest versions of Android and iOS both for application and for web application running on common browsers. The code we used to sample the gyro via a web page can be found in Appendix B. The results indicate that a Gecko based browser does not limit the sampling frequency beyond the limit imposed by the operating system, while WebKit and Blink based browsers does impose stricter limits on it.

2.3.2 Aliasing

As noted above, the sampling frequency of a gyro is uniform and can be at most 200 Hz. This allows us to directly sense audio signals of up to 100 Hz. Aliasing is a phenomenon where for a sinusoid of frequency f, sampled with frequency f_s, the resulting samples are indistinguishable from those of another sinusoid of frequency $|f - N \cdot f_s|$, for any integer N. The values corresponding to $N \neq 0$ are called images or aliases of frequency f. An undesirable phenomenon in general, here aliasing allows us to sense audio signals having frequencies which are higher than 100 Hz, thereby extracting more information from the gyroscope readings. This is illustrated in Figure 3.

Using the gyro, we recorded a single 280 Hz tone. Figure 3(a) depicts the recorded signal in the frequency domain (x-axis) over time (y-axis). A lighter shade in the spectrogram indicates a stronger signal at the corresponding frequency and time values. It can be clearly seen that there is a strong signal sensed at frequency 80 Hz starting around 1.5 sec. This is an alias of the 280 Hz-tone. Note that the aliased tone is indistinguishable from an actual tone at the aliased frequency. Figure 3(b) depicts a recording of multiple short tones between 130 Hz and 200 Hz. Again, a strong signal can be seen at the aliased frequencies corresponding to 130 - 170 Hz[3]. We also observe some weaker aliases that do not correspond to the base frequencies of the recorded tones, and perhaps correspond to their harmonics. Figure 3(c) depicts the recording of a chirp in the range of 420 - 480 Hz. The aliased chirp is detectable in the range of 20 - 80 Hz; however it is a rather weak signal.

[3]We do not see the aliases corresponding to 180 - 200 Hz, which might be masked by the noise at low frequencies, i.e., under 20 Hz.

2.3.3 Self noise

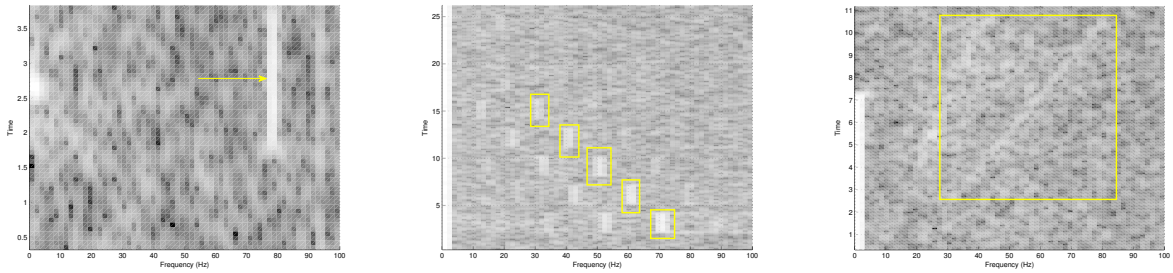
The self noise characteristic of a microphone indicates what is the most quiet sound, in decibels, a microphone can pick up, i.e. the sound that is just over its self noise. To measure the gyroscope's self noise we played 80 Hz tones for 10 seconds at different volumes while measuring it using a decibel meter. Each tone was recorded by the Galaxy S III gyroscope. While analyzing the gyro recordings we realized that the gyro readings have a noticeable increase in amplitude when playing tones with volume of 75 dB or higher which is comparable to the volume of a loud conversation. Moreover, a FFT plot of the gyroscope recordings gives a noticeable peak at the tone's frequency when playing tone with a volume as low as 57 dB which is below the sound level of a normal conversation. These findings indicate that a gyro can pick up audio signals which are lower than 100 HZ during most conversations made over or next to the phone. To test the self noise of the gyro for aliased tones we played 150 Hz and 250 Hz tones. The lowest level of sound the gyro picked up was 67 dB and 77 dB, respectively. These are much higher values that are comparable to a loud conversation.

2.3.4 Directionality

We now measure how the angle at which the audio signal hits the phone affects the gyro. For this experiment we played an 80 Hz tone at the same volume three times. The tone was recorded at each time by the Galaxy S III gyro while the phone rested at a different orientation allowing the signal to hit it parallel to one of its three axes (see Figure 4). The gyroscope senses in three axes, hence for each measurement the gyro actually outputs three readings – one per axis. As we show next this property benefits the gyro's ability to pick up audio signals from every direction. For each recording we calculated the FFT magnitude at 80 Hz. Table 2 summarizes the results.

It is obvious from the table that for each direction the audio hit the gyro, there is at least one axis whose readings are dominant by an order of magnitude compared to the rest. This can be explained by STMicroelectronics gyroscope design as depicted in Figure 1[4]. When the signal travels in parallel to the phone's x or y axes, the sound pressure vibrates mostly masses laid along the respective axis, i.e. M_2 and M_4 for x axis and M_1 and M_3

[4]This is the design of the gyro built into Galaxy S III.



(a) A single 280 Hz tone (b) Multiple tones in the range of 130 – 170 Hz (c) A chirp in the range of 420 – 480 Hz

Figure 3: Example of aliasing on a mobile device. Nexus 4 (a,c) and Galaxy SII (b).

Tone direction:	X			Y			Z		
Recording direction:	x	y	z	x	y	z	x	y	z
Amplitude:	0.002	0.012	0.0024	0.01	0.007	0.004	0.007	0.0036	0.0003

Table 2: Sensed amplitude for every direction of a tone played at different orientations relative to the phone. For each orientation the dominant sensed directions are emphasized.

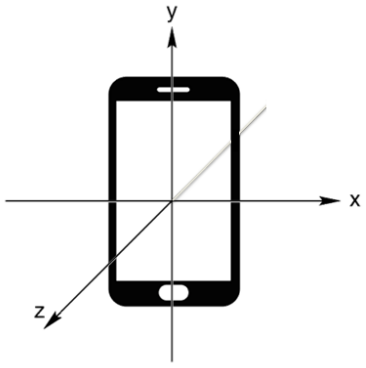


Figure 4: Coordinate system of Android and iOS.

for the y axis; therefore, the gyro primarily senses a rotation at the y or x axes, respectively (see Section 2.1.1). When the signal travels in parallel to the phone's z axis then the sound pressure vibrates all the 4 masses up and down, hence the gyro primarily senses a rotation at both x and y axes.

These findings indicate that the gyro is an omni-directional audio sensor allowing it to pick up audio signal from every direction.

3 Speech analysis based on a single gyroscope

In this section we show that the acoustic signal measured by a single gyroscope is sufficient to extract information about the speech signal, such as speaker characteristics and identity, and even recognize the spoken words or phrases. We do so by leveraging the fact that aliasing causes information leaks from higher frequency bands into the sub-Nyquist range.

Since the fundamentals of human voices are roughly in the range of 80 – 1100 Hz [20], we can capture a large fraction of the interesting frequencies, considering the results we observe in 2.3.2. Although we do not delve into comparing performance for different types of speakers, one might expect that given a stronger gyroscope response for low frequencies, typical adult male speech (Bass, Baritone, Tenor) could be better analyzed than typical female or child speech (Alto, Mezzo-Soprano, Soprano) [5], however our tests show that this is not necessarily the case.

The signal recording, as captured by the gyroscope, is not comprehensible to a human ear, and exhibits a mixture of low frequencies and aliases of frequencies beyond the Nyquist sampling frequency (which is 1/2 the sampling rate of the Gyroscope, i.e. 100 Hz). While the signal recorded by a single device does not resemble speech, it is possible to train a machine to transcribe the signal with significant success.

Speech recognition tasks can be classified into several types according to the setup. Speech recognition can handle fluent speech or isolated words (or phrases); operate on a closed set of words (finite dictionary) or an open set[6]; It can also be speaker dependent (in which case the recognizer is trained per speaker) or speaker in-

[5] For more information about vocal range see `http://www.wikipedia.org/wiki/Vocal_range`

[6] For example by identifying phonemes and combining them to words.

dependent (in which case the recognizer is expected to identify phrases pronounced by different speakers and possibly ones that were not encountered in the training set). Additionally, speech analysis may be also used to identify the speaker.

We focused on speaker identification (including gender identification of the speaker) and isolated words recognition while attempting both speaker independent and speaker dependent recognition. Although we do not demonstrate fluent speech transcription, we suggest that successful isolated words recognition could be fairly easily transformed into a transcription algorithm by incorporating word slicing and HMM [40]. We did not aim to implement a state-of-the-art speech recognition algorithm, nor to thoroughly evaluate or do a comparative analysis of the classification tests. Instead, we tried to indicate the potential risk by showing significant success rates of our speech analysis algorithms compared to randomly guessing. This section describes speech analysis techniques that are common in practice, our approach, and suggestions for further improvements upon it.

3.1 Speech processing: features and algorithms

3.1.1 Features

It is common for various feature extraction methods to view speech as a process that is stationary for short time windows. Therefore speech processing usually involves segmentation of the signal to short (10 – 30 ms) overlapping or non-overlapping windows and operation on them. This results in a time-series of features that characterize the time-dependent behavior of the signal. If we are interested in time-independent properties we shall use spectral features or the statistics of those time-series (such as mean, variance, skewness and kurtosis).

Mel-Frequency Cepstral Coefficients (MFCC) are widely used features in audio and speech processing applications. The Mel-scale basically compensates for the non-linear frequency response of the human ear[7]. The Cepstrum transformation is an attempt to separate the excitation signal originated by air passing through the vocal tract from the effect of the vocal tract (acting as a filter shaping that excitation signal). The latter is more important for the analysis of the vocal signal. It is also common to take the first and second derivatives of the MFCC as additional features, indicative of temporal changes [30].

Short Time Fourier Transform (STFT) is essentially a spectrogram of the signal. Windowing is applied to short overlapping segments of the signal and FFT is computed. The result captures both spectral and time-dependent features of the signal.

[7] Approximated as logarithmic by the Mel-scale

3.1.2 Classifiers

Support Vector Machine (SVM) is a general binary classifier, trained to distinguish to groups. We use SVM to distinguish male and female speakers. Multi-class SVMs can be constructed using multiple binary SVMs, to distinguish between multiple groups. We used a multi-class SVM to distinguish between multiple speakers, and to recognize words from a limited dictionary.

Gaussian Mixture Model (GMM) has been successfully used for speaker identification [41]. We can train a GMM for each group in the training stage. In the testing stage we can obtain a match score for the sample using each one of the GMMs and classify the sample according to the group corresponding to the GMM that yields the maximum score.

Dynamic Time Warping (DTW) is a time-series matching and alignment technique [37]. It can be used to match time-dependent features in presence of misalignment or when the series are of different lengths. One of the challenges in word recognition is that the samples may differ in length, resulting in different number of segments used to extract features.

3.2 Speaker identification algorithm

Prior to processing we converted the gyroscope recordings to audio files in WAV format while upsampling them to 8 KHz[8]. We applied silence removal to include only relevant information and minimize noise. The silence removal algorithm was based on the implementation in [29], which classifies the speech into voiced and unvoiced segments (filtering out the unvoiced) according to dynamically set thresholds for Short-Time Energy and Spectral Centroid features computed on short segments of the speech signal. Note that the gyroscope's zero-offset yields particularly noisy recordings even during unvoiced segments.

We used statistical features based on the first 13 MFCC computed on 40 sub-bands. For each MFCC we computed the mean and standard deviation. Those features reflect the spectral properties which are independent of the pronounced word. We also use delta-MFCC (the derivatives of the MFCC), RMS Energy and

[8] Although upsampling the signal from 200 Hz to 8 KHz does not increase the accuracy of audio signal, it is more convenient to handle the WAV file at higher sampling rate with standard speech processing tools.

Spectral Centroid statistical features. We used MIRToolbox [32] for the feature computation. It is important to note that while MFCC have a physical meaning for real speech signal, in our case of an narrow-band aliased signal, MFCC don't necessarily have an advantage, and were used partially because of availability in MIRToolbox. We attempted to identify the gender of the speaker, distinguish between different speakers of the same gender and distinguish between different speakers in a mixed set of male and female speakers. For gender identification we used a binary SVM, and for speaker identification we used multi-class SVM and GMM. We also attempted gender and speaker recognition using DTW with STFT features. All STFT features were computed with a window of 512 samples which, for sampling rate of 8 KHz, corresponds to 64 ms.

3.3 Speech recognition algorithm

The preprocessing stage for speech recognition is the same as for speaker identification. Silence removal is particularly important here, as the noisy unvoiced segments can confuse the algorithm, by increasing similarity with irrelevant samples. For word recognition, we are less interested in the spectral statistical features, but rather in the development of the features in time, and therefore suitable features could be obtained by taking the full spectrogram. In the classification stage we extract the same features for a sample y. For each possible label l we obtain a similarity score of the y with each sample X_i^l corresponding to that guess in the training set. Let us denote this similarity function by $D(y, X_i^l)$. Since different samples of the same word can differ in length, we use DTW. We sum the similarities to obtain a total score for that guess

$$S^l = \sum_i D(y, X_i^l)$$

After obtaining a total score for all possible words, the sample is classified according to the maximum total score

$$C(y) = \operatorname*{argmax}_l S^l$$

3.4 Experiment setup

Our setup consisted of a set of loudspeakers that included a sub-woofer and two tweeters (depicted in Figure 5). The sub-woofer was particularly important for experimenting with low-frequency tones below 200 Hz. The playback was done at volume of approximately 75 dB to obtain as high SNR as possible for our experiments. This means that for more restrictive attack scenarios (farther source, lower volume) there will be a need to handle low SNR, perhaps by filtering out the noise or applying some other preprocessing for emphasizing the speech signal. [9]

Figure 5: Experimental setup

3.4.1 Data

Due to the low sampling frequency of the gyro, a recognition of speaker-independent general speech would be an ambitious long-term task. Therefore, in this work we set out to recognize speech of a limited dictionary, the recognition of which would still leak substantial private information. For this work we chose to focus on the digits dictionary, which includes the words: zero, one, two..., nine, and "oh". Recognition of such words would enable an attacker to eavesdrop on private information, such as credit card numbers, telephone numbers, social security numbers and the like. This information may be eavesdropped when the victim speaks over or next to the phone.

In our experiments, we use the following corpus of audio signals on which we tested our recognition algorithms.

TIDIGITS This is a subset of a corpus published in [33]. It includes speech of isolated digits, i.e., 11 words per speaker where each speaker recorded each word twice. There are 10 speakers (5 female and 5 male). In total, there are $10 \times 11 \times 2 = 220$ recordings. The corpus is digitized at 20 kHz.

3.4.2 Mobile devices

We primarily conducted our experiments using the following mobile devices:

[9] We tried recording in an anechoic chamber, but it didn't seem to provide better recognition results compared to a regular room. We therefore did not proceed with the anechoic chamber experiments. Yet, further testing is needed to understand whether we can benefit significantly from an anechoic environment.

1. Nexus 4 phone which according to a teardown analysis [13] is equipped with an InvenSense MPU-6050 [12] gyroscope and accelerometer chip.

2. Nexus 7 tablet which according to a teardown analysis [14] is equipped with an InverSense MPU-6050 gyroscope and accelerometer.

3. Samsung Galaxy S III phone which according to a teardown analysis [6] is equipped with an STMicroelectronics LSM330DLC [10] gyroscope and accelerometer chip.

3.5 Sphinx

We first try to recognize digit pronunciations using general-purpose speech recognition software. We used Sphinx-4 [47] – a well-known open-source speech recognizer and trainer developed in Carnegie Mellon University. Our aim for Sphinx is to recognize gyro-recordings of the TIDIGITS corpus. As a first step, in order to test the waters, instead of using actual gyro recordings we downsampled the recordings of the TIDITS corpus to 200 Hz; then we trained Sphinx based on the modified recordings. The aim of this experiment is to understand whether Sphinx detects any useful information from the sub-100 Hz band of human speech. Sphinx had a reasonable success rate, recognizing about 40% of pronunciations.

Encouraged by the above experiment we then recorded the TIDIGITS corpus using a gyro – both for Galaxy S III and Nexus 4. Since Sphinx accepts recording in WAV format we had to convert the raw gyro recordings. Note that at this point for each gyro recording we had 3 WAV files, one for each gyro axis. The final stage is silence removal. Then we trained Sphinx to create a model based on a training subset of the TIDIGITS, and tested it using the complement of this subset.

The recognition rates for either axes and either Nexus 4 or Galaxy S III were rather poor: 14% on average. This presents only marginal improvement over the expected success of a random guess which would be 9%.

This poor result can be explained by the fact that Sphinx's recognition algorithms are geared towards standard speech recognition tasks where most of the voice-band is present and is less suited to speech with very low sampling frequency.

3.6 Custom recognition algorithms

In this section we present the results obtained using our custom algorithm. Based on the TIDIGITS corpus we randomly performed a 10-fold cross-validation. We refer mainly to the results obtained using Nexus 4 gyroscope readings in our discussion. We also included in the tables some results obtained using a Galaxy III device, for comparison.

	SVM	GMM	DTW
Nexus 4	80%	72%	84%
Galaxy S III	82%	68%	58%

Table 3: Speaker's gender identification results

		SVM	GMM	DTW
Nexus 4	Mixed female/male	23%	21%	50%
	Female speakers	33%	32%	45%
	Male speakers	38%	26%	65%
Galaxy S III	Mixed female/male	20%	19%	17%
	Female speakers	30%	20%	29%
	Male speakers	32%	21%	25%

Table 4: Speaker identification results

Results for gender identification are presented in Table 3. As we see, using DTW scoring for STFT features yielded a much better success rate.

Results for speaker identification are presented in Table 4. Since the results for a mixed female-male set of speakers may be partially attributed to successful gender identification, we tested classification for speakers of the same gender. In this setup we have 5 different speakers. The improved classification rate (except for DTW for female speaker set) can be partially attributed to a smaller number of speakers.

The results for speaker-independent isolated word recognition are summarized in Table 5. We had correct classification rate of $\sim 10\%$ using multi-class SVM and GMM trained with MFCC statistical features, which is almost equivalent to a random guess. Using DTW with STFT features we got 23% correct classification for male speakers, 26% for female speakers and 17% for a mixed set of both female and male speakers. The confusion matrix in Figure 6, corresponding to the mixed speaker-set recorded on a Nexus 4, explains the not so high recognition rate, exhibiting many false positives for the words "6" and "9". At the same time the recognition rate for

		SVM	GMM	DTW
Nexus 4	Mixed female/male	10%	9%	17%
	Female speakers	10%	9%	26%
	Male speakers	10%	10%	23%
Galaxy S III	Mixed female/male	7%	12%	7%
	Female speakers	10%	10%	12%
	Male speakers	10%	6%	7%

Table 5: Speaker-independent case – isolated words recognition results

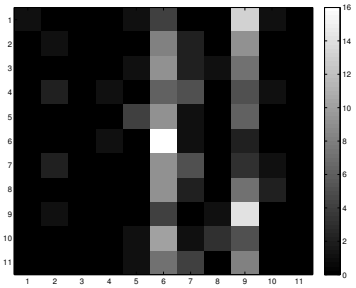


Figure 6: Speaker independent word recognition using DTW: confusion matrix as a heat map. $c_{(i,j)}$ corresponds to the number of samples from group i that were classified as j, where i, j are the row and column indices respectively.

SVM	GMM	DTW
15%	5%	65%

Table 6: Speaker-dependent case – isolated words recognition for a single speaker. Results obtained via "leave-one-out" cross-validation on 44 recorded words pronounced by a single speaker. Recorded using a Nexus 4 device.

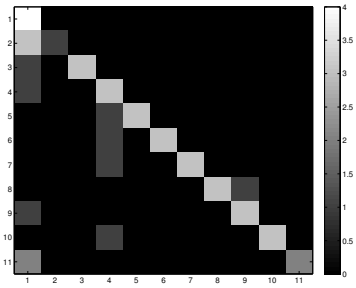


Figure 7: Speaker dependent word recognition using DTW: confusion matrix as a heat map.

these particular words is high, contributing to the correct identification rate.

For a speaker-dependent case one may expect to get better recognition results. We recorded a set of 44 digit pronunciations, where each digit was pronounced 4 times. We tested the performance of our classifiers using "leave-one-out" cross-validation. The results are presented in Table 6, and as we expected exhibit an improvement compared to the speaker independent recognition[10] (except for GMM performance that is equivalent to randomly guessing). The confusion matrix corresponding to the word recognition in a mixed speaker-set using DTW is presented in Figure 7.

DTW method outperforms SVM and GMM in most cases. One would expect that DTW would perform better for word recognition since the changing in time of the spectral features is taken into account. While true for Nexus 4 devices it did not hold for measurements taken with Galaxy III. possible explanation to that is that the low-pass filtering on the Galaxy III device renders all methods quite ineffective resulting in a success rate equivalent to a random guess. For gender and speaker identification, we would expect statistical spectral features based methods (SVM and GMM) to perform at least as good as DTW. It is only true for the Galaxy S III mixed speaker set and gender identification cases, but not for the other experiments. Specifically for gender identification, capturng the temporal development of the spectral feature wouldn't seem like a clear advantage and is therefore somewhat surprising. One comparative study that supports the advantage of DTW over SVM for speaker recognition is [48]. It doesn't explain though why it outperforms GMM which is a well established method for speaker identification. More experimentation is required to confirm whether this phenomenon is consistent and whether it is related to capturing the high frequencies.

3.7 Further improvement

We suggest several possible future improvements on our recognition algorithms. Phoneme recognition instead of whole words, in combination with an HMM could improve the recognition results. This could be more suitable since different pronunciations have different lengths, while an HMM could introduce a better probabilistic recognition of the words. Pre-filtering of the signal could be beneficial and reduce irrelevant noise. It is not clear which frequencies should be filtered and therefore some experimentation is needed to determine it.

[10] It is the place to mention that a larger training set for speaker independent word recognition is likely to yield better results. For our tests we used relatively small training and evaluation sets.

For our experiments, we used samples recorded by the gyroscope for training. For speaker-dependent speech recognition we can imagine it may be easier to obtain regular speech samples for a particular speaker than a transcribed recording of gyroscope samples. Even for speaker independent speech recognition, it would be easier to use existing audio corpora for training a speech recognition engine than to produce gyroscope recordings for a large set of words. For that purpose it would be interesting to test how well the recognition can perform when the training set is based on normal audio recordings, downsampled to 200 Hz to simulate a gyroscope recording.

Another possible improvement is to leverage the 3-axis recordings. It is obvious that the three recordings are correlated while the noise of gyro readings is not. Hence, one may take advantage of this to get a composed signal of the three axes to get a better signal-to-noise ratio.

While we suggested that the signal components related to speech, and those related to motion lie in separate frequency bands, the performance of speech analysis in the presence of such noise is yet to be evaluated.

4 Reconstruction using multiple devices

In this section we suggest that isolated word recognition can be improved if we sample the gyroscopes of multiple devices that are in close proximity, such that they exhibit a similar response to the acoustic signals around them. This can happen for instance in a conference room where two mobile devices are running malicious applications or, having a browser supporting high-rate sampling of the gyroscope, are tricked into browsing to a malicious website.

We do not refer here to the possibility of using several different gyroscope readings to effectively obtain a larger feature vector, or have the classification algorithm take into account the score obtained for all readings. While such methods to exploit the presence of more than one acoustic side-channel may prove very efficient we leave them outside the scope of this study. It also makes sense to look into existing methods for enhancing speech recognition using multiple microphones, covered in signal processing and machine learning literature (e.g., [23]).

Instead, we look at the possibility of obtaining an enhanced signal by using all of the samples for reconstruction, thus effectively obtaining higher sampling rate. Moreover, we hint at the more ambitious task of reconstructing a signal adequate enough to be comprehensible by a human listener, in a case where we gain access to readings from several compromised devices. While there are several practical obstacles to it, we outline the idea, and demonstrate how partial implementation of it facilitates the automatic speech recognition task.

We can look at our system as an array of time-interleaved data converters (interleaved ADCs). Interleaved ADCs are multiple sampling devices where each samples the signal with a sub-Nyquist frequency. While the ADCs should ideally have time offsets corresponding to a uniform sampling grid (which would allow to simply interleave the samples and reconstruct according to the Whittaker-Shannon interpolation formula [44]), usually there will be small time skews. Also, DC offsets and different input gains can affect the result and must all be compensated.

This problem is studied in a context of analog design and motivated by the need to sample high-frequency signals using low-cost and energy-efficient low-frequency A/D converters. While many papers on the subject exist, such as [27], the proposed algorithms are usually very hardware centric, oriented towards real-time processing at high-speed, and mostly capable of compensating for very small skews. Some of them require one ADC that samples the signal above the Nyquist rate, which is not available in our case. At the same time, we do not aim for a very efficient, real-time algorithm. Utilizing recordings from multiple devices implies offline processing of the recordings, and we can afford a long run-time for the task.

The ADCs in our case have the same sampling rate $F_s = 1/T = 200$. We assume the time-skews between them are random in the range $[0, T_Q]$ where for N ADCs $T_Q = \frac{T}{N}$ is the Nyquist sampling period. Being located at different distances from the acoustic source they are likely to exhibit considerably different input gains, and possibly have some DC offset. [26] provides background for understanding the problems arising in this configuration and covers some possible solutions.

4.1 Reconstruction algorithm

4.1.1 Signal offset correction

To correct a constant offset we can take the mean of the Gyro samples and compare it to 0 to get the constant offset. It is essentially a simple DC component removal.

4.1.2 Gain mismatch correction

Gain mismatch correction is crucial for a successful signal reconstruction. We correct the gain by normalizing the signal to have standard deviation equal to 1. In case we are provided with some reference signal with a known peak, we can adjust the gains of the recordings so that the amplitude at this peak is equal for all of them.

4.1.3 Time mismatch correction

While gyroscope motion events are provided with precise timestamps set by the hardware, which theoretically could have been used for aligning the recordings, in practice, we cannot rely on the clocks of the mobile devices to be synchronized. Even if we take the trouble of synchronizing the mobile device clock via NTP, or even better, a GPS clock, the delays introduced by the network, operating system and further clock-drift will stand in the way of having clock accuracy on the order of a millisecond[11]. While not enough by itself, such synchronization is still useful for coarse alignment of the samples.

El-Manar describes foreground and background time-mismatch calibration techniques in his thesis [27]. Foreground calibration means there is a known signal used to synchronized all the ADCs. While for the purpose of testing we can align the recordings by maximizing the cross-correlation with a known signal, played before we start recording, in an actual attack scenario we probably won't be able to use such a marker[12]. Nevertheless, in our tests we attempted aligning using a reference signal as well. It did not exhibit a clear advantage over obtaining coarse alignment by finding the maximum of the cross-correlation between the signals. One can also exhaustively search a certain range of possible offsets, choosing the one that results in a reconstruction of a sensible audio signal.

Since this only yields alignment on the order of a sampling period of a single gyroscope (T), we still need to find the more precise time-skews in the range $[0,T]$. We can scan a range of possible time-skews, choosing the one that yields a sensible audio signal. We can think of an automated evaluation of the result by a speech recognition engine or scoring according to features that would indicate human speech, suggesting a successful reconstruction.

This scanning is obviously time consuming. If we have n sources, we set one of the time skews (arbitrary) to 0, and have $n-1$ degrees of freedom to play with, and the complexity grows exponentially with the number of sources. Nevertheless, in an attack scenario, it is not impossible to manually scan all possibilities looking for the best signal reconstruction, provided the information is valuable to the eavesdropper.

[11] Each device samples with a period of 5 ms, therefore even 1 ms clock accuracy would be quite coarse.

[12] While an attacker may be able to play using one of the phones' speakers a known tone/chirp (no special permissions are needed), it is unlikely to be loud enough to be picked up well by the other device, and definitely depends on many factors such as distance, position etc.

4.1.4 Signal reconstruction from non-uniform samples

Assuming we have compensated for offset, gain mismatch and found the precise time-skews between the sampling devices, we are dealing with the problem of signal reconstruction from periodic, non-uniform samples. A seminal paper on the subject is [28] by Eldar et al. Among other works in the field are [39, 46] and [31]. Sindhi et al. [45] propose a discrete time implementation of [28] using digital filterbanks. The general goal is, given samples on a non-uniform periodic grid, to obtain estimation of the values on a uniform sampling grid, as close as possible to the original signal.

A theoretic feasibility justification lies in Papoulis' Generalized Sampling theorem [38]. Its corollary is that a signal bandlimited to π/T_Q can be recovered from the samples of N filters with sampling periods $T = NT_Q$. [13] We suggest using one of the proposed methods for signal reconstruction from periodic non-uniform samples. With only several devices the reconstructed speech will still be narrow-band. While it won't necessarily be easily understandable by a human listener, it could be used for better automated identification. Applying narrowband to wideband speech extension algorithms [36] might provide audio signals understandable to a human listener.

We suggest using one of the methods for signal reconstruction from periodic non-uniform samples mentioned above. With only several devices the reconstructed speech will still be narrow-band. For example, using readings from two devices operating at 200 Hz and given their relative time-skew we obtain an effective sampling rate of 400 Hz. For four devices we obtain a sampling rate of 800 Hz, and so on. While a signal reconstructed using two devices still won't be easily understandable by a human listener, it could be used to improve automatic identification.

We used [28] as a basis for our reconstruction algorithm. The discussion of *recurrent non-uniform sampling* directly pertains to our task. It proposes a filterbank scheme to interpolate the samples such that an approximation of the values on the uniform grid is obtained. The derivation of the discrete-time interpolation filters is provided in Appendix A.

This method allows us to perform reconstruction with arbitrary time-skews; however we do not have at the time a good method for either a very precise estimation

[13] It is important to note that in our case the signal is not necessarily bandlimited as required. While the base pitch of the speech can lie in the range $[0, 200 \cdot N]$, it can contain higher frequencies that are captured in the recording due to aliasing, and may interfere with the reconstruction. It depends mainly on the low-pass filtering applied by the gyroscope. In InvenSense's MPU-6050, Digital Low-Pass Filtering (DLPF) is configurable through hardware registers [11], so the conditions depend to some extent on the particular driver implementation.

SVM	GMM	DTW
18%	14%	77%

Table 7: Evaluation of the method of reconstruction from multiple devices. Results obtained via "leave-one-out" cross-validation on 44 recorded words pronounced by a single speaker. Recorded using a Nexus 4 device.

of the time-skews or automatic evaluation of the reconstruction outcome (which would enable searching over a range of possible values). For our experiment we applied this method to the same set of samples used for speaker-dependent speech recognition evaluation, which was recorded simultaneously by two devices. We used the same value for τ, the time-skew for all samples, and therefore chose the expected value $\tau = T/2$ which is equivalent to the particular case of sampling on a uniform grid (resulting in all-pass interpolation filters). It is essentially the same as interleaving the samples from the two readings, and we ended up implementing this trivial method as well, in order to avoid the adverse effects of applying finite non-ideal filters.

It is important to note that while we propose a method rooted in signal processing theory, we cannot confidently attribute the improved performance to obtaining a signal that better resembles the original, until we take full advantage of the method by estimating the precise time-skew for each recording, and applying true non-uniform reconstruction. It is currently left as an interesting future improvement, for which the outlined method can serve as a starting point. In this sense, our actual experiment can be seen as taking advantage of better feature vectors, comprised of data from multiple sources.

4.1.5 Evaluation

We evaluated this approach by repeating the speaker-dependent word recognition experiment on signals reconstructed from readings of two Nexus 4 devices. Table 7 summarizes the final results obtained using the sample interleaving method[14].

There was a consistent noticeable improvement compared to the results obtained using readings from a single device, which supports the value of utilizing multiple gyroscopes. We can expect that adding more devices to the setup would further improve the speech recognition.

[14] We also compared the performance of the DTW classifier on samples reconstructed using the filterbank approach. It yielded a slightly lower correct classification rate of 75% which we attribute to the mentioned effects of applying non-ideal finite filters.

5 Further Attacks

In this section we suggest directions for further exploitation of the gyroscopes:

Increasing the gyro's sampling rate. One possible attack is related to the hardware characteristics of the gyro devices. The hardware upper bound on sampling frequency is higher than that imposed by the operating system or by applications[15]. InvenSense MPU-6000/MPU-6050 gyroscopes can provide a sampling rate of up to 8000 Hz. That is the equivalent of a POTS (telephony) line. STMicroelectronics gyroscopes only allow up to 800 Hz sampling rate, which is still considerably higher than the 200 Hz allowed by the operating system (see Appendix C). If the attacker can gain a one-time privileged access to the device, she could patch an application, or a kernel driver, thus increasing this upper bound. The next steps of the attack are similar: obtaining gyroscope measurements using an application or tricking the user into leaving the browser open on some website. Obtaining such a high sampling rate would enable using the gyroscope as a microphone in the full sense of hearing the surrounding sounds.

Source separation. Based on experiments' results presented in Section 2.3.4 it is obvious that the gyro's measurements are sensitive to the relative direction from which the acoustic signal arrives. This may give rise to the possibility to detect the angle of arrival (AoA) at which the audio signal hits the phone. Using AoA detection one may be able to better separate and process multiple sources of audio, e.g. multiple speakers near the phone.

Ambient sound recognition. There are works (e.g. [42]) which aim to identify a user's context and whereabouts based on the ambient noise detected by his smart phone, e.g restaurant, street, office, and so on. Some contexts are loud enough and may have distinct fingerprint in the low frequency range to be able to detect them using a gyroscope, for example railway station, shopping mall, highway, and bus. This may allow an attacker to leak more information on the victim user by gaining indications of the user's whereabouts.

6 Defenses

Let us discuss some ways to mitigate the potential risks. As it is often the case, a secure design would require an

[15] As we have shown, the sampling rate available on certain browsers is much lower than the maximum sampling rate enabled by the OS. However, this is an application level constraint.

overall consideration of the whole system and a clear definition of the power of the attacker against whom we defend. To defend against an attacker that has only user-level access to the device (an application or a website), it might be enough to apply low-pass filtering to the raw samples provided by the gyroscope. Judging by the sampling rate available for Blink and WebKit based browsers, it is enough to pass frequencies in the range 0 – 20 Hz. If this rate is enough for most of the applications, the filtering can be done by the driver or the OS, subverting any attempt to eavesdrop on higher frequencies that reveal information about surrounding sounds. In case a certain application requires an unusually high sampling rate, it should appear in the list of permissions requested by that application, or require an explicit authorization by the user. To defend against attackers who gain root access, this kind of filtering should be performed at the hardware level, not being subject to configuration. Of course, it imposes a restriction on the sample rate available to applications.

Another possible solution is some kind of acoustic masking. It can be applied around the sensor only, or possibly on the case of the mobile device.

7 Conclusion

We show that the acoustic signal measured by the gyroscope can reveal private information about the phone's environment such as who is speaking in the room and, to some extent, what is being said. We use signal processing and machine learning to analyze speech from very low frequency samples. With further work on low-frequency signal processing of this type it should be possible to further increase the quality of the information extracted from the gyro.

This work demonstrates an unexpected threat resulting from the unmitigated access to the gyro: applications and active web content running on the phone can eavesdrop sound signals, including speech, in the vicinity of the phone. We described several mitigation strategies. Some are backwards compatible for all but a very small number of applications and can be adopted by mobile hardware vendors to block this threat.

A general conclusion we suggest following this work is that access to all sensors should be controlled by the permissions framework, possibly differentiating between low and high sampling rates.

Acknowledgements

We would like to thank Nimrod Peleg, from the Signal and Image Processing Lab (SIPL) at the Technion, for providing assistance with the TIDIGITS corpus. We also greatful to Sanjay Kumar Sindhi, from IIT Madras, for providing implementation and testing of several signal reconstruction algorithms. We would also like to thank Prof. Jared Tanner, from UC Davis, and Prof. Yonina Eldar, from the Technion, for advising on reconstruction of non-uniformly sampled signals. We thank Hriso Bojinov for taking part in the initial brainstorming related to this research and finally, Katharina Roesler, for proofreading and advising on writing and formulation.

This work was supported by NSF and the DARPA SAFER program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or DARPA.

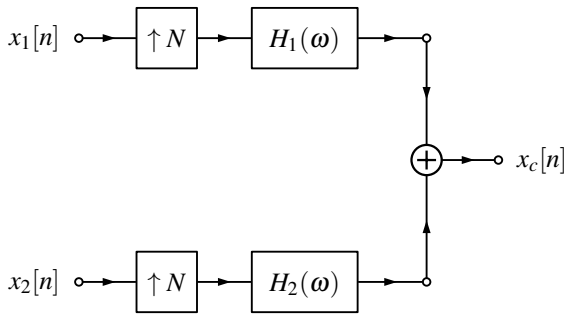


Figure 8: Filterbank reconstruction scheme

A Signal reconstruction from Recurrent Non-Uniform Samples

Here we present the derivation of the discrete-time interpolation filters used in our implementation. The notation in the expressions corresponds to the notation in [28]. The continuous time expression for the interpolation filters according to Eq. 18 in [28] is given by

$$h_p(t) = a_p sinc\left(\frac{t}{T}\right) \prod_{q=0, q\neq p}^{N-1} \sin\left(\frac{\pi\left(t+t_p-t_q\right)}{T}\right)$$

We then sample this expression at times $t = nT_Q - t_p$ and calculate the filter coefficients for 48 taps. Given these filters, the reconstruction process consists of upsampling the input signals by factor N, where $N = T/T_Q$ is the number of ADCs, filtering and summation of the outputs of all filters (as shown in Figure 8).

B Code for sampling a gyroscope via a HTML web-page

For a web page to sample a gyro the DeviceMotion class needs to be utilized. In the following we included a JavaScript snippet that illustrates this:

```
if(window.DeviceMotionEvent) {
  window.addEventListener('devicemotion', function(
      event) {
    var r = event.rotationRate;
    if ( r!=null ) {
      console.log('Rotation at [x,y,z] is: [' +
        r.alpha+','+r.beta+','+r.gamma+']\n');
    }
  }
}
```

Figure 9 depicts measurements of the above code running on Firefox (Android) while sampling an audio chirp 50 – 100 Hz.

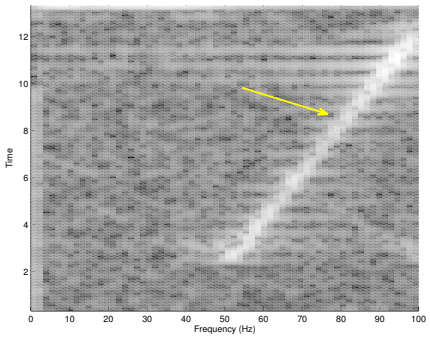


Figure 9: Recording audio at 200 Hz using JavaScript code on a web-page accessed from the Firefox browser for Android.

C Gyroscope rate limitation on Android

Here we see a code snippet from the Invensense driver for Android, taken from *hardware/invensense/65xx/libsensors_iio/MPLSensor.cpp*. The OS is enforcing a rate of 200 Hz.

```
static int hertz_request = 200;
#define DEFAULT_MPL_GYRO_RATE          (20000L)      //us
...
#define DEFAULT_HW_GYRO_RATE           (100)         //Hz
#define DEFAULT_HW_ACCEL_RATE          (20)          //ms
...
/* convert ns to hardware units */
#define HW_GYRO_RATE_NS                (1000000000LL / rate_request) // to Hz
#define HW_ACCEL_RATE_NS               (rate_request / (1000000L))   // to ms
...
/* convert Hz to hardware units */
#define HW_GYRO_RATE_HZ                (hertz_request)
#define HW_ACCEL_RATE_HZ               (1000 / hertz_request)
```

D Code Release

We provide the source code of the Android application we used for recording the sensor measurements, as well as the Matlab code we used for analyzing the data and training and testing of the speech recognition algorithms. We also provide the gyroscope recordings used for the evaluation of our method. The code and data can be downloaded from the project website at `http://crypto.stanford.edu/gyrophone`. In addition, we provide a web page that records gyroscope measurements if accessed from a device that supports it.

References

[1] 3-axis digital gyroscopes. `http://www.st.com/st-web-ui/static/active/en/resource/sales_and_marketing/promotional_material/flyer/fl3axdigitalgyro.pdf`.

[2] Corona SDK API reference. `http://docs.coronalabs.com/api/library/system/setGyroscopeInterval.html`.

[3] Galaxy Tab 7.7. `http://www.techrepublic.com/blog/cracking-open/galaxy-tab-77-teardown-reveals-lots-of-samsungs-homegrown-hardware/588/`.

[4] Inside the Latest Galaxy Note 3. `http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-galaxy-note-3/`.

[5] Inside the Samsung Galaxy S4. `http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-samsung-galaxy-s4/`.

[6] Inside the Samsung Galaxy SIII. `http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-samsung-galaxy-siii/`.

[7] InvenSense Inc. `http://www.invensense.com/`.

[8] iPad Mini Retina Display Teardown. `http://www.ifixit.com/Teardown/iPad+Mini+Retina+Display+Teardown/19374`.

[9] L3G4200D data sheet. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00265057.pdf`.

[10] LSM330DLC data sheet. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037200.pdf`.

[11] MPU-6000 and MPU-6050 Register Map and Descriptions. `http://invensense.com/mems/gyro/documents/RM-MPU-6000A.pdf`.

[12] MPU-6050 product specification. `http://www.invensense.com/mems/gyro/documents/PS-MPU-6000A-00v3.4.pdf`.

[13] Nexus 4 Teardown. `http://www.ifixit.com/Teardown/Nexus+4+Teardown/11781`.

[14] Nexus 7 Teardown. `http://www.ifixit.com/Teardown/Nexus+7+Teardown/9623`.

[15] STMicroelectronics Inc. `http://www.st.com/`.

[16] Everything about STMicroelectronics 3-axis digital MEMS gyroscopes. `http://www.st.com/web/en/resource/technical/document/technical_article/DM00034730.pdf`, July 2011.

[17] iPhone 5S MEMS Gyroscope STMicroelectronics 3x3mm - Reverse Costing Analysis. `http://www.researchandmarkets.com/research/lxrnrn/iphone_5s_mems`, October 2013.

[18] MEMS for Cell Phones and Tablets. `http://www.i-micronews.com/upload/Rapports/Yole_MEMS_for_Mobile_June_2013_Report_Sample.pdf`, July 2013.

[19] AL-HAIQI, A., ISMAIL, M., AND NORDIN, R. On the best sensor for keystrokes inference attack on android. *Procedia Technology 11* (2013), 989–995.

[20] APPELMAN, D. *The Science of Vocal Pedagogy: Theory and Application*. Midland book. Indiana University Press, 1967.

[21] CAI, L., AND CHEN, H. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security* (2011), USENIX Association, pp. 9–9.

[22] CASTRO, S., DEAN, R., ROTH, G., FLOWERS, G. T., AND GRANTHAM, B. Influence of acoustic noise on the dynamic performance of mems gyroscopes. In *ASME 2007 International Mechanical Engineering Congress and Exposition* (2007), pp. 1825–1831.

[23] CRAMMER, K., KEARNS, M., AND WORTMAN, J. Learning from multiple sources. *The Journal of Machine Learning Research 9* (2008), 1757–1774.

[24] DEAN, R. N., CASTRO, S. T., FLOWERS, G. T., ROTH, G., AHMED, A., HODEL, A. S., GRANTHAM, B. E., BITTLE, D. A., AND BRUNSCH, J. P. A characterization of the performance of a mems gyroscope in acoustically harsh environments. *Industrial Electronics, IEEE Transactions on 58*, 7 (2011), 2591–2596.

[25] DEAN, R. N., FLOWERS, G. T., HODEL, A. S., ROTH, G., CASTRO, S., ZHOU, R., MOREIRA, A., AHMED, A., RIFKI, R., GRANTHAM, B. E., ET AL. On the degradation of mems gyroscope performance in the presence of high power acoustic noise. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on* (2007), IEEE, pp. 1435–1440.

[26] EL-CHAMMAS, M., AND MURMANN, B. *Background calibration of time-interleaved data converters*. Springer, 2012.

[27] EL-CHAMMAS, M. I. *Background Calibration of Timing Skew in Time-Interleaved A/D Converters*. Stanford University, 2010.

[28] ELDAR, Y. C., AND OPPENHEIM, A. V. Filterbank reconstruction of bandlimited signals from nonuniform and generalized samples. *Signal Processing, IEEE Transactions on 48*, 10 (2000), 2864–2875.

[29] GIANNAKOPOULOS, T. A method for silence removal and segmentation of speech signals, implemented in matlab.

[30] HASAN, M. R., JAMIL, M., AND RAHMAN, M. G. R. M. S. Speaker identification using mel frequency cepstral coefficients. *variations 1* (2004), 4.

[31] JOHANSSON, H. K., AND LÖWENBORG, P. *Reconstruction of periodically nonuniformly sampled bandlimited signals using time-varying FIR filters*. 2005.

[32] LARTILLOT, O., TOIVIAINEN, P., AND EEROLA, T. A matlab toolbox for music information retrieval. In *Data analysis, machine learning and applications*. Springer, 2008, pp. 261–268.

[33] LEONARD, R. G., AND DODDINGTON, G. TIDIGITS. `http://catalog.ldc.upenn.edu/LDC93S10`, 1993.

[34] MANTYJARVI, J., LINDHOLM, M., VILDJIOUNAITE, E., MAKELA, S.-M., AND AILISTO, H. Identifying users of portable devices from gait pattern with accelerometers. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on* (2005), vol. 2, IEEE, pp. ii–973.

[35] MARQUARDT, P., VERMA, A., CARTER, H., AND TRAYNOR, P. (sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 551–562.

[36] MIET, G. Towards wideband speech by narrowband speech bandwidth extension: magic effect or wideband recovery? *These de doctorat, University of Maine* (2001).

[37] MÜLLER, M. Dynamic time warping. *Information retrieval for music and motion* (2007), 69–84.

[38] PAPOULIS, A. Generalized sampling expansion. *Circuits and Systems, IEEE Transactions on 24*, 11 (1977), 652–654.

[39] PRENDERGAST, R. S., LEVY, B. C., AND HURST, P. J. Reconstruction of band-limited periodic nonuniformly sampled signals through multirate filter banks. *Circuits and Systems I: Regular Papers, IEEE Transactions on 51*, 8 (2004), 1612–1622.

[40] RABINER, L. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* (1989).

[41] REYNOLDS, D. A., AND ROSE, R. C. Robust text-independent speaker identification using gaussian mixture speaker models. *Speech and Audio Processing, IEEE Transactions on 3*, 1 (1995), 72–83.

[42] ROSSI, M., FEESE, S., AMFT, O., BRAUNE, N., MARTIS, S., AND TROSTER, G. Ambientsense: A real-time ambient sound recognition system for smartphones. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on* (2013), IEEE, pp. 230–235.

[43] SEEGER, J., LIM, M., AND NASIRI, S. Development of high-performance high-volume consumer MEMS gyroscopes. `http://www.invensense.com/mems/gyro/documents/whitepapers/Development-of-High-Performance-High-Volume-Consumer-MEMS-Gyroscopes.pdf`.

[44] SHANNON, C. E. Communication in the presence of noise. *Proceedings of the IRE 37*, 1 (1949), 10–21.

[45] SINDHI, S., AND PRABHU, K. Reconstruction of N-th Order Nonuniformly Sampled Signals Using Digital Filter Banks. *commsp.ee.ic.ac.uk* (2012).

[46] STROHMER, T., AND TANNER, J. Fast reconstruction algorithms for periodic nonuniform sampling with applications to time-interleaved adcs. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on* (2007), vol. 3, IEEE, pp. III–881.

[47] WALKER, W., LAMERE, P., KWOK, P., RAJ, B., SINGH, R., GOUVEA, E., WOLF, P., AND WOELFEL, J. Sphinx-4: A flexible open source framework for speech recognition. Tech. rep., 2004.

[48] YEE, L., AND AHMAD, A. Comparative Study of Speaker Recognition Methods: DTW, GMM and SVM. *comp.utm.my*.