POLITECNICO
MILANO 1863

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Doctoral Programme In Computer Science and Engineering

Defending from Financially-Motivated Software Abuses

Doctoral Dissertation of:
Andrea Continella

Advisor:
Prof. Stefano Zanero

Co-Advisor:
Federico Maggi

Tutor:
Prof. Andrea Bonarini

Chair of the Doctoral Program:
Prof. Andrea Bonarini

2017 – XXX

Abstract

Software is involved in every aspect of our world, from our homes to large enterprises, and, in particular, it manages our data. As a consequence, software abuses can drastically impact our lives, for instance causing substantial financial losses or affecting people's privacy. This raised the attention of cybercriminals, who found in this scenario a lucrative business. In fact, in the past twenty years the motivation behind the cybercriminals' modus operandi has changed. No longer searching only for notoriety and fame, they have turned their attention to financial gain. Indeed malicious software, "malware," is one of the most dangerous Internet threat nowadays.

This dissertation details our research on the analysis and detection of the current software abuses, with the aim of protecting users from such threats. Specifically, we focus on three main threats, which have been the cause of billion dollars losses in the past years. First, we concentrate on information-stealing malware, also known as "banking Trojans." The purpose of these Trojans is to steal banking credentials and any other kind of private information by loading code in memory and hooking the network-related operating-system APIs used by web browsers. Second, we focus on a major class of malware, known as ransomware, which encrypts files, preventing legitimate access until a ransom is paid. Finally, we analyze the privacy issues in the mobile world by studying the problem of privacy leaks. Mobile apps are notorious for collecting a wealth of private information from users. Such information is particularly attractive. For instance, cybercriminals are known to sell users' private information on the underground markets, and advertisement libraries massively collect users' data to illicitly increase their profits.

Our contributions regarding banking Trojans focus on extracting robust, behavioral signatures of their malicious behavior, by combining web-page differential analysis and memory forensics techniques. The produced signatures can then be used, on the client side, to detect such Trojans in a more generic way, independently from their specific implementation, and protect victims' machines.

Our contributions regarding ransomware focus on designing behavioral detection models and proposing a novel defense mechanism to mitigate its effectiveness by equipping modern operating systems with practical self-healing capabilities. We designed our detection models after an analysis of billions of low-level, I/O filesystem requests generated by thousands of benign applications, which we collected from clean machines in use by real users for about one month.

Our contributions regarding mobile privacy leaks focus on proposing a novel, obfuscation-resilient approach to detect privacy leaks by applying network differential analysis. To make differential analysis practical, our approach leverages a novel technique that performs root cause analysis of non-determinism in the network behavior of Android apps.

Sommario

Oggigiorno il software è coinvolto in ogni aspetto del nostro mondo, dalle nostre case alle grandi industrie, ed, in particolare, gestisce i nostri dati. Di conseguenza, abusi del software possono avere un impatto drastico sulle nostre vite, ad esempio causando ingenti perdite economiche o violando la privacy delle persone. Tutto ciò ha attirato l'attenzione dei cybercriminali, che hanno trovato in questo scenario un business lucrativo; negli ultimi venti anni le motivazioni del modus operandi dei cybercriminali sono cambiate, infatti essi non cercano più solamente notorietà e fama, ma hanno rivolto la loro attenzione a guadagni illeciti. Difatti oggi il software malevolo, "*malware*," è una delle minacce più pericolose di Internet.

Questa tesi illustra le nostre ricerche sull'analisi e la rilevazione degli attuali abusi del software, allo scopo di proteggere gli utenti da tali minacce. In particolare, ci concentriamo su tre principali minacce, che sono state la causa principale di perdite di miliardi di dollari negli ultimi anni. Innanzitutto, ci concentriamo su *information-stealing malware*, noti anche come "*banking Trojans*." L'obiettivo di questi Trojans è quello di rubare le credenziali bancarie e qualsiasi altro tipo di informazione privata caricando codice malevolo in memoria ed intercettando le chiamate alle API di rete utilizzate dai browser web. In secondo luogo, ci concentriamo su una grande classe di malware, conosciuta come "*ransomware*," che cifra i file impedendone l'accesso legittimo fino al momento in cui non viene pagato un riscatto. Infine, analizziamo le problematiche della privacy nel mondo mobile studiando il problema dei *privacy leaks*. Le app mobile sono note per raccogliere un gran numero di informazioni riservate degli utenti; tali informazioni hanno, dal punto di vista economico, un grandissimo valore. Ad esempio, i criminali informatici sono noti per vendere informazioni private degli utenti sui mercati *underground*, e le librerie pubblicitarie raccolgono massicciamente tali dati degli utenti per incrementare illecitamente i loro profitti.

I nostri contributi riguardo i banking Trojans si concentrano sull'estrazione di robuste *signature* comportamentali, combinando *web-page differential analysis* con tecniche di ispezione forense della memoria. Le signature prodotte possono quindi essere utilizzate, sul lato client, per individuare tali Trojan in un modo più generico ed indipendentemente dalla loro specifica implementazione, così da proteggere le macchine degli utenti.

I nostri contributi relativi a ransomware si concentrano sulla progettazione di modelli di rilevamento comportamentali e di un nuovo meccanismo di difesa per mitigare l'efficacia di questi attacchi; per fare ciò abbiamo dotato i moderni sistemi operativi di funzionalità di auto-guarigione (*self-healing*). Abbiamo progettato i nostri modelli di rilevazione dopo un'analisi di miliardi di operazioni I/O a basso livello sul filesystem, generate da migliaia di applicazioni benigne, che abbiamo raccolto da macchine pulite in uso da parte di utenti reali per circa un mese.

I nostri contributi su mobile privacy leak si concentrano sulla proposta di un nuovo approccio per rilevare privacy leak in modo resiliente a tecniche di offuscamento, applicando *network differential analysis*. Per rendere pratico l'uso della differential analysis il nostro approccio sfrutta una nuova tecnica che esegue un'analisi delle cause principali di non-determinismo nel comportamento di rete delle applicazioni Android.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
AV	Anti-Virus
C&C	Command and Control
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
DLL	Dynamic Loaded library
DNS	Domain Name System
DOM	Document Object Model
FS	File System
HTML	Hypertext Markup Language
HTTP(S)	Hypertext Transfer Protocol (Secure)
IP	Internet Protocol
OS	Operating System
OTP	One-Time Password
PE	Portable Executable
PID	Process Identifier
PIN	Personal Identification Number
REGEX	Regular Expression
TEB	Thread Environment Block
TIB	Thread Information Block
URL	Uniform Resource Locator
VM	Virtual Machine
XML	eXtensible Markup Language

1. Introduction

As a consequence of the continuous advances in technology, computers run nowadays most of the fundamental tasks of our society, and software controls such devices and their operations. Most importantly, people's personal data are entirely managed by software programs. However, in parallel to the diffusion of connected digital devices we have witnessed an enormous rise of cybercrime. In fact, criminals found in this digital world a lucrative business. As a matter of fact, financially-motivated malware is nowadays one of the most dangerous Internet threat.

The concept of computer malware was introduced in 1987 by Cohen [21], who theorized the possibility for a computer program to perform malicious actions without users' control, and to spread across connected machines. Since then, malicious software has widely evolved. Initially, malware authors developed malicious programs just to prove their skills and talent. For instance, the Morris worm in 1988 caused a massive denial of service of computers connected to the Internet. The attack was just an experiment and there was no financial motivation behind its development: *"The goal of this program was to demonstrate the inadequacies of current security measures on computer networks by exploiting the security defects that Morris had discovered."*[1] Instead, today malware authors aim at gaining profits, implementing behaviors that range from stealing online banking credentials to locking or encrypting users' files in order to ask for a ransom payment. Moreover, there is an active ecosystem behind these criminal activities, which, like an industry, provides malware toolkits that can be purchased online and configured through easy-to-use user interfaces.

This scenario is further amplified by the huge amount of data that users produce everyday. According to Cisco [20], the average amount of Internet IP traffic reached 96,054 PB per month in 2016 and Internet reached 3,424,971,237 users [2]. This data is extremely attractive for cybercriminals, who built a lucrative ecosystem that leverages users' information to make illicit profits. For instance, Figure 1.1 shows the common fraud scheme behind the sophisticated money stealing process. The first step is the malware implementation. Malware authors implement malware toolkits and put them on sale on the underground markets. Second, cybercriminals buy a toolkit, create a customized sample (or the malware authors themselves create the executable), and start spreading it to infect victims. Then, the stolen money are kept on bank accounts that are not in the criminals' name, but they are property of another actor, called "money mule." Money mules receive the stolen money, keep part of the sum for themselves and move the rest to the criminals' real accounts. In this way, cybercriminals add another layer between themselves and the victims, making it very hard to identify the real responsible behind the fraud [32].

[1] US v. Morris, 928 F. 2d 504 - Court of Appeals, 2nd Circuit 1991

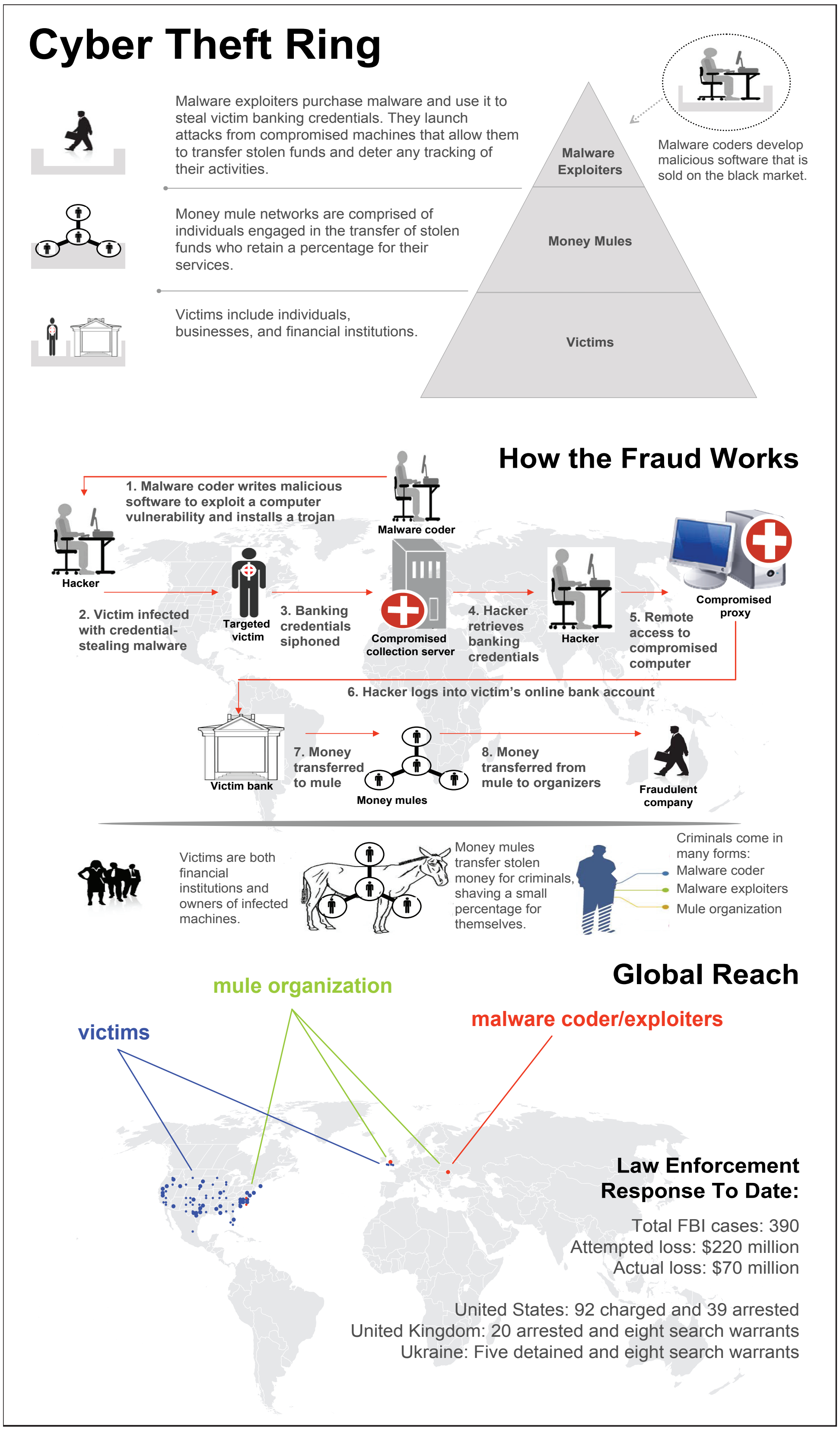

Cyber Theft Ring
Malware exploiters purchase malware and use it to steal victim banking credentials. They launch attacks from compromised machines that allow them to transfer stolen funds and deter any tracking of their activities.
Money mule networks are comprised of individuals engaged in the transfer of stolen funds who retain a percentage for their services.
Victims include individuals, businesses, and financial institutions.
Malware Exploiters
Money Mules
Victims
Malware coders develop malicious software that is sold on the black market.
How the Fraud Works
1. Malware coder writes malicious software to exploit a computer vulnerability and installs a trojan
Malware coder
Hacker
2. Victim infected with credential-stealing malware
Targeted victim
3. Banking credentials siphoned
Compromised collection server
4. Hacker retrieves banking credentials
Hacker
5. Remote access to compromised computer
Compromised proxy
6. Hacker logs into victim's online bank account
Victim bank
7. Money transferred to mule
Money mules
8. Money transferred from mule to organizers
Fraudulent company
Victims are both financial institutions and owners of infected machines.
Money mules transfer stolen money for criminals, shaving a small percentage for themselves.
Criminals come in many forms:
Malware coder
Malware exploiters
Mule organization
Global Reach
mule organization
victims
malware coder/exploiters
Law Enforcement Response To Date:
Total FBI cases: 390
Attempted loss: $220 million
Actual loss: $70 million
United States: 92 charged and 39 arrested
United Kingdom: 20 arrested and eight search warrants
Ukraine: Five detained and eight search warrants

Figure 1.1: Common cyber banking fraud scheme (source [32] 2010).

In this dissertation, we detail our research on the analysis and detection of the current software abuses, with the aim of defending users from such threats. In particular, we concentrate on three main threats, which caused billion dollars losses in the past years. First, we focus on information-stealing malware, also known as "banking Trojans," a class of malware that steals victims' private information (e.g., banking credentials) by taking control of the victims' browser —*Man in the Browser* attacks— in order to perform financial frauds. Second, we focus on ransomware, another class of malware that encrypts victims' files, preventing legitimate access until a ransom is paid. Third, we focus on mobile privacy leaks. Mobile apps collect a wealth of users' private information, which is particularly attractive. In fact, cybercriminals are known to sell users' private information on the underground markets, and advertisement libraries massively gather such data to illicitly increase their profits.

1.1 Todays' Security Threats

Every year, new threats are discovered and, while attackers take advantage of them until effective countermeasures are found, researchers and security experts continuously implement new defense mechanism to protect users.

Symantec detected 357 millions of new malware variants in 2016 [89]. The number of financial Trojan detections decreased by 36 percent in 2016 (73 percent in 2015) [98] and mobile banking malware targeted more than 170 apps for credential stealing.

Extortion-based schemes turned out to be particularly effective for cybercriminals in the last years. Ransomware, malware that encrypts users' files and asks for a ransom to release the decryption key(s), has been the most prevalent class of malware in the last two years. From 2015 to 2016, the number of ransomware families increased from 30 to 100, and the average ransom amount raised from 294 USD to 1,077 USD [89]. Such a great diffusion made experts define 2016 as the "year of extortion."

Furthermore, the last years have been also dominated by high-profile data breaches. In the last 8 years more than 7.1 billion identities have been exposed in data breaches [89]. For instance, in July 2017, Equifax, a consumer credit score company, revealed unauthorized access up to 143 million customer account details, including names, social security numbers, drivers licenses, and credit card numbers of around 200,000 people [54]. This proves how cybercriminals are interested in users' private data. This is due to the fact that people's information are particularly profitable on the underground markets, which today run a very proficient business: everyone can buy credit card information, full identities, or rent a scam hosting solution.

1.2 Original Contributions

In the aforementioned threat landscape, our main research area focuses on financially-motivated software abuses. In particular, we focus on generic approaches to detect these malicious activities and protect users from such threats. Since todays' threats are complex and continuously evolving, we need our solutions to be generic and to adapt themselves to the changes quickly. Our contributions focus on the mitigation of three main threats that have been widely spread and caused billion dollars losses: banking Trojans, ransomware, mobile privacy leaks.

1.2.1 Banking Trojans Analysis and Detection

Banking Trojans can be detected by static signatures that precisely identify malicious binaries. However, this approach is not generic and strongly depends on the implementation details of the malware sample. In addition, new families and new versions of such Trojans are constantly released, and each specific sample can be customized and obfuscated, generating new, distinct executables. For these reasons, we propose a novel, generic, and effective approach to analyze and detect the common behavior of this malware. Modern Trojans are in fact equipped with a common functionality, called *WebInject*, used by cybercriminals to silently modify web pages on the infected hosts.

In summary:

- We proposed a tool, Prometheus, that, based on web-page differential analysis, characterizes WebInject mechanisms in an implementation-independent fashion, without needing a-priori knowledge about the API hooking method, nor on the specific configuration encryption-decryption mechanisms used by the malware. Our approach generates robust, behavioral signatures of the WebInject behavior.
- We combined the web page differential analysis with a memory forensics inspection technique to validate the generated signatures.
- We performed experiments on a dataset of real, active Trojans, and provided insights from a data analysis point of view (i.e., classification of the URLs where injections occur typically) that is used for validating our approach.
- We developed a prototype tool, Iris, that leverages the signatures produced by Prometheus to check, on the client side, whether a web page is rendered on an infected machine. Our tool works by inspecting the memory of running browser processes and reconstructing the DOMs of the visited web pages to match injection signatures and spot artifacts of malicious web-injections.

1.2.2 Protection from Ransomware Attacks

Preventive and reactive security measures can only partially mitigate the damage caused by modern ransomware attacks. In fact, pure-detection approaches (e.g., based on analysis sandboxes or pipelines) are not sufficient, because, when luck allows a sample to be isolated and analyzed, it is already too late for several users. We believe that a forward-looking solution is to equip modern operating systems with generic, practical self-healing capabilities against this serious threat.

In summary:

- We performed the first, large-scale data collection of I/O request packets generated by benign applications in real-world conditions. Our dataset contains about 1.7 billion IRPs produced by 2,245 different applications.
- We proposed a ransomware-detection approach that enables a modern operating system to recognize the typical signs of ransomware behaviors.
- We proposed an approach that makes a modern filesystem resilient to malicious encryption, by dynamically reverting the effects of ransomware attacks.

- We implemented these approaches in ShieldFS as a drop-in, Windows kernel module that we showed capable of successfully protecting from current ransomware attacks.

1.2.3 Mobile Privacy Leaks Detection

Despite significant effort from the research community in developing privacy leak detection tools based on data flow tracking inside the app or through network traffic analysis, it is still unclear whether apps and ad libraries can hide the fact that they are leaking private information. In fact, all existing analysis tools have limitations: data flow tracking suffers from imprecisions that cause false positives, as well as false negatives when the data flow from a source of private information to a network sink is interrupted; on the other hand, network traffic analysis cannot handle encryption or custom encoding. We propose a new approach to privacy leak detection that is not affected by such limitations, and it is also resilient to obfuscation techniques, such as encoding, formatting, encryption, or any other kind of transformation performed on private information before it is leaked.

In summary:

- We developed a tool, Agrigento, that performs root cause analysis of non-determinism in the network behavior of Android apps.
- We showed that, in most cases, non-determinism in network behavior can be explained and eliminated. This key insight makes privacy leak detection through differential black-box analysis practical.
- The results of our empirical study provide new insights into how modern apps use custom encoding and obfuscation techniques to stealthily leak private information and to evade existing approaches.

The aforementioned results have been published in the proceedings of international conferences and international journals.

1.3 Document Structure

This document is structured as follows. Chapter 2 focuses on banking Trojans. Specifically, after an initial description of the details of this kind of malware, and a review of the research works proposed in the state-of-the-art around this topic, we present our novel analysis approach and the results of our experiments published in [23].

In Chapter 3, we focus on ransomware and propose our approach to protect users against this threat. Specifically, we initially present the results of our preliminary study on the filesystem activity. Then, based on the obtained results, we details our detection and protection methodology published in [22]. Finally, we describe our experimental evaluation.

Chapter 4 focuses on mobile privacy leak detection. We first describe the challenges in applying black-box differential analysis at the network level, then we present the techniques we proposed to address such challenges and the approach we published in [24]. Last, we evaluate our system prototype.

Finally, Chapter 5 wraps up the results of our research by summarizing the key points and setting new challenges for future works.

2. Analyzing and Detecting WebInject-based Information Stealers

Nowadays malware is reaching high levels of sophistication. The number of families and variants observed increased exponentially in the last years. A particular type of Trojans, known as Information-stealers or banking Trojans, allows malware operators to intercept sensitive data such as credentials (e.g., usernames, passwords) and credit cards information. Such malware uses a Man-in-the-Browser technique that infects a web browser by hooking certain functions in libraries and modifying web pages.

More precisely, they use two main attack techniques: (1) modification of network Windows APIs (2) and modification of web pages requested by the web client for specific websites (e.g., banks). The first technique intercepts at network API level any sensitive data that are processed by the browser even in case the connection is encrypted. The second technique typically adds new form fields to the web pages, in order to steal target information such as One-Time Passwords. Each attack module relies on an encrypted configuration file that contains a list of targeted URLs (e.g., well-known banks URLs) in form of regular expressions along with the HTML/JavaScript code that should be injected to a particular website. Given its flexibility, WebInject-based malware has become a popular information-stealing mechanism nowadays [97].

Furthermore, these Trojans are sold on underground markets along with automatic frameworks that include web-based administration panels, builders and customization procedures.

Different works have been done regarding the analysis and detection of banking Trojans [14], [18], [30], [67], [70], [75] but, as explained in §2.2.1, most of them are dependent on a specific malware family or version, and require a considerable effort to be constantly adapted to new emerging techniques. We aim to precisely characterize WebInject behaviors without relying on the implementation details of a malware but mainly on its own injection behavior.

More precisely, based on findings of our previous work [25], which demonstrated the feasibility of web page differential analysis, we propose Prometheus, an automatic framework for analyzing banking Trojans. The proposed system works independently from the implementation details of the malware and the key idea is based on the fact that actions of malware must eventually result in changing the document object model (DOM). Comparing DOMs downloaded in clean machines with those downloaded in infected machines allows us to generate signatures and extract the WebInject configuration file. Our approach also exploits the artifacts left in memory by malware during the injection process and uses them for validating the results of the DOMs analysis. It is important to note that our detection mechanism exploits a different angle of malware behavior and it can be deployed as an additional detection layer along with others approaches [85, 31, 55] (e.g., AVs, IDS etc.). Our system is orthogonal to any other detection techniques and offers another protection layer that can be used to improve the detection capabilities.

We evaluated Prometheus on a dataset of 135 distinct samples of ZeuS analyzing 68 real, live URLs of banking websites. We manually verified the results of our experiments and we show that Prometheus is able to generate signatures correctly with a negligible fraction (0.70%) of "false differences," which are those legitimate differences wrongly detected as malicious injections. Our experiments also show the efficiency of our system, which is able to analyze one URL every 6 seconds on average.

2.1 Background on Information Stealers

Information-stealing Trojans is a growing [106], sophisticated threat. The most famous example is ZeuS, from which other descendants were created. This malware is actually a binary generator, which eases the creation of customized variants. For instance, as of February 23, 2016, according to ZeuS Tracker[1], there are 8,149 distinct variants that have yet to be included in the Malware Hash Registry database[2]. Notice that this is an underestimation, limited to binaries that are currently tracked. This high number of variants results in a low detection rate overall (40.05% as of February 23, 2016).

Lindorfer et al. [60] measured that Trojans such as ZeuS and GenericTrojan are actively developed and maintained. These and other modern malware families live in a complex environment with development kits, web-based administration panels, builders, automated distribution networks, and easy-to- use customization procedures. The most alarming consequence is that virtually anyone can buy a malware builder from underground marketplaces and create a customized sample. Lindorfer et al. [60] also found an interesting development evolution, which indicates a need for forward-looking malware-analysis methods that are less dependent on the current or past characteristics of malware samples or families. This also relates to the fact that the source code is sometimes leaked (e.g., CARBERP, ZeuS), which leads to further creation of new variants [86].

2.1.1 The underground economy

One of the banking Trojan problems is that anyone, independently from their skill level, can perform financial frauds, as the underground marketplace is active and provides all the required resources, like a service industry. For example, Goncharov [41] estimated for the only Russian underground economy a 2.3 billion dollars market.

Grier et al. [43] investigated the emergence of the exploit- as-a-service model, showing how attackers pay for exploit kits to infect victims and propagate their own malware through drive-by downloads. Therefore, even with little or no expertise or ability to write a malware, anyone can simply purchase these "kits," and follow detailed guides and video tutorials sold online. The Trojan samples and services available on the underground markets vary, and their price depends on the features. Typically, it starts from 100$ for an old, leaked version, to about 3,000$ for a new, complete version [86]. Furthermore, cybercriminals offer paid support and customization, or sell

[1] `https://zeustracker.abuse.ch/statistic.php` [2] `http://www.team-cymru.org/Services/MHR/`

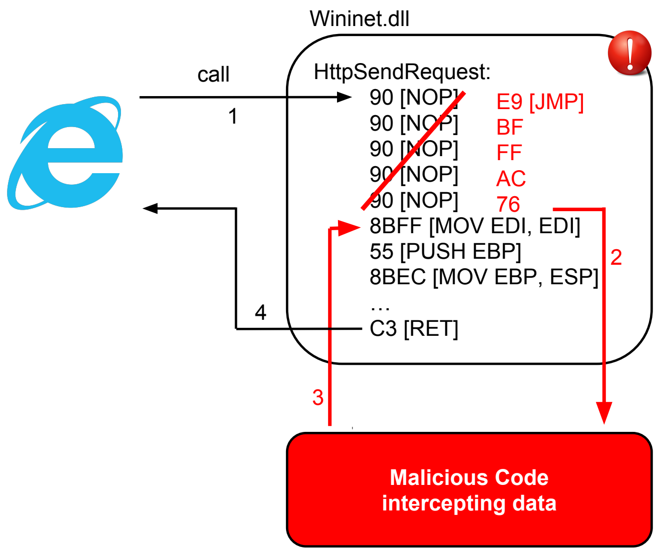


Figure 2.1: Example of API hooking.

advanced configuration files that the end users can include in their custom builds. Custom WebInjects can be also purchased for 30-100 [97].

2.1.2 Man in the Browser attacks and WebInject

Financial Trojans use Man-in-the-Browser (MitB) techniques to perform attacks. These techniques exploit API hooking and, as the name suggests, allow malware to be logically executed inside the web browser and to intercept all data flowing through it. Also, modern banking Trojan families commonly include a module called WebInject [97], which facilitates the manipulation and modification of data transmitted between a web server and the browser. Once the victim is infected, the WebInject module places itself between the browser's rendering engine and the API networking functions used for sending and receiving data. By hooking high-level API communication functions in user-mode code, the Trojans can intercept data more conveniently than traditional keyloggers, as they can intercept data after being decrypted. Therefore, the WebInject module is effective even in case an HTTPS connection is used. Figure 2.2 shows a high level view of the injection mechanism, and an example of inline hooking, in which the Trojan overwrites the first five bytes of the `HttpSendRequest` function with a jump to malicious code. Exploiting the high level configuration interface of the WebInject module, cybercriminals can effectively inject HTML code that adds extra fields in forms so as to steal sensitive information. The goal is to make the victim believe that the web page is legitimately asking for a second factor of authentication or other sensitive information (as illustrated in Figure 2.3). In fact, the victim will notice no suspicious signs (e.g., invalid SSL certificate or different URL) because the page is modified "on the fly" right before being displayed, directly on the local machine.

The WebInject module loads an encrypted configuration file. This file contains the list of WebInject rules, which include the target URLs, or regular expressions that match more than a single URL, and the HTML/JavaScript code to be injected into specific web pages. For each rule, the attackers can set two hooks (`data_before` and `data_after`) that identify the portion of the web page where the new content, defined

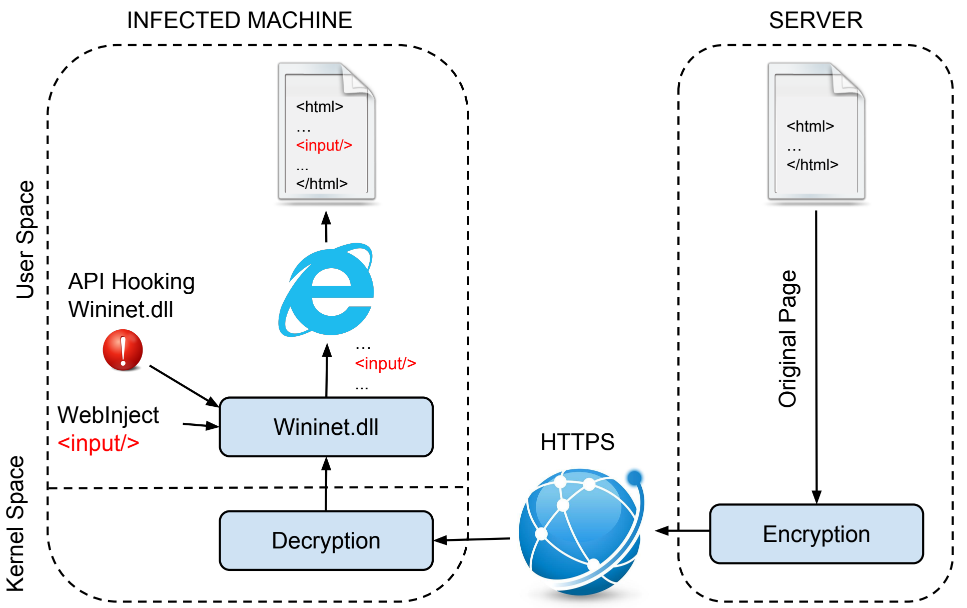
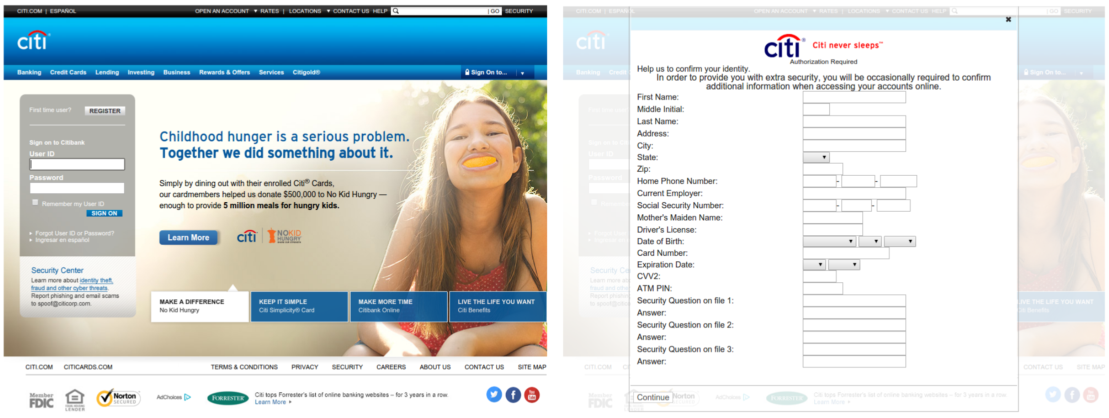


Figure 2.2: High level view of the Injection mechanism

by the `data_inject` variable, is injected. An example of a real WebInject rule is shown in Figure 2.4.

2.2 State of the Art and Research Challenges

New families and new versions of info-stealing Trojan samples are frequently released [15, 88, 99] and each specific Trojan can be customized and obfuscated, generating new, distinct executables. In addition, the custom configuration files are encrypted and embedded in the final executable. For these reasons, manually analyzing all the samples is not scalable. Thus, automatic mechanisms to extract valuable information from encrypted configuration files or for analyzing the activity of an infected machine are needed.



Childhood hunger is a serious problem.
Together we did something about it.

Simply by dining out with their enrolled Citi® Cards, our cardmembers helped us donate $500,000 to No Kid Hungry — enough to provide **5 million meals for hungry kids.**

Security Center
Learn more about identity theft, fraud and other cyber threats.
Report phishing and email scams to spoof@citicorp.com.

Citi tops Forrester's list of online banking websites – for 3 years in a row.
Learn More ▸

Authorization Required

Help us to confirm your identity.
In order to provide you with extra security, you will be occasionally required to confirm additional information when accessing your accounts online.

First Name:
Middle Initial:
Last Name:
Address:
City:
State:
Zip:
Home Phone Number:
Current Employer:
Social Security Number:
Mother's Maiden Name:
Driver's License:
Date of Birth:
Card Number:
Expiration Date:
CVV2:
ATM PIN:
Security Question on file 1:
Answer:
Security Question on file 2:
Answer:
Security Question on file 3:
Answer:

Figure 2.3: Example of a real injection on the home page of `online.citibank.com`.

2.2.1 Information Stealers Analysis and Detection

Bruescher et al. [18] proposed an approach to identify WebInject-based information stealers. The idea is similar to the typical rootkit-detection approach based on recognizing the presence of API hooks in common loaded libraries, especially in browser and Internet-related APIs. To avoid false positives (e.g., legitimate hooks), they inspect the destination of each hook, and check if the pointed module is trusted and correctly signed. The main limitation of this detection approach is the strong dependence on the version of the Trojan, on the operating system, and on the hooked browser. Different Trojans or future releases could change the list of API functions to hook, or target another browser that uses different libraries. We argue that, in general, user-level API hooking is not the only method to achieve MitB functionalities.

In Heiderich et al. [45] the authors protect the browser from malicious websites that dynamically change the DOM. Although not designed specifically to target information stealers, such mechanism could be applied for recognizing WebInjects. In details, their system instruments the ECMA script layer by proxying its functions. However, as the authors mention, their method can only detect changes of the DOM that occur at runtime, whereas WebInjects work at the source-code level. This means that they are not able to identify modifications (e.g., node insertion) not performed via JavaScript code.

Wyke et al. [100] outlines a sandbox-based system that automatically analyzes banking Trojans observing the network traffic, and extracting valuable information such as command and control addresses, in a scalable and extensible way. Although this approach still needs to be manually updated to support newer malware versions, it is a solid tool that can be used to complement our approach.

2.3 Approach Overview

We divide our approach into two parts: web page differential analysis and memory forensic analysis.

First, we assume that a page rendered on an infected machine includes the injected portions of code. This is reasonable and realistic, as WebInjects-based Trojans need to perform code injections. In contrast, the same page rendered on a “clean” machine contains the original source code. Hence, our approach consists in analyzing informa-

```
set_url *.wellsfargo.com/* G
  data_before
    <span class="mozcloak"><input type="password"*</span>
  data_end
  data_inject
    <br><strong><label for="atmpin">ATM PIN</label>:</strong>&nbsp;
    <br /><span class="mozcloak"><input type="password" accesskey="A"
    id="atmpin" name="USpass" size="13" maxlength="14" style="width:
    147px" tabindex="2" /></span>
  data_end
  data_after
  data_end
```

Figure 2.4: Example of a real WebInject rule

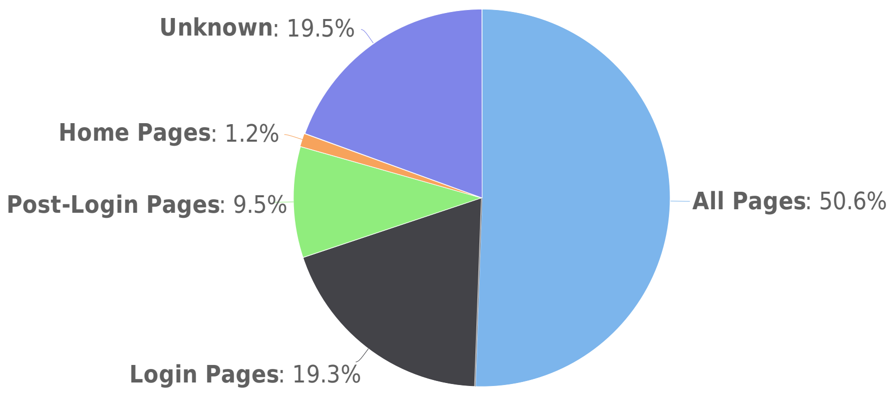


Figure 2.5: Classification of the 694 extracted regexes according to the kind of pages targeted.

tion stealers looking for evidence of injections in the web pages, and extracting this evidence through a comparison of the web pages with the original ones. This method does not leverage any malware-specific component or vulnerability to observe and characterize the injection behavior, therefore it is more generic by design. However, it could generate many false differences since the content of a web page may vary legitimately. For example, this could be due to server-side script, or advertisements that include dynamically changing content. To cope with this, we collect many versions of the same web page, and compare them to discard legitimate differences that occur between two or more of them. Moreover, to reduce false differences, we designed four heuristic-based filters (i.e., whitelisting clean differences, ignoring node or attribute reordering, filtering harmless differences, identifying repeated malicious injections). As further discussed in §4.6, although filtering legitimate changes creates an opportunity for the attacker to hide in such differences, this is unlikely to happen and easy to remediate collaborating with banks and receiving real-time updates about changes in the web pages.

Secondly, we assume that, during the injection process, Trojans leave artifacts of their configuration files, for examples list of targets URLs, attributes, etc. in memory. Our approach is to recover such artifacts from the memory of an infected machine, through memory forensic techniques that inspect the memory of the target applications, and search for strings tokens associated to the definition of regular expressions, part of any WebInject configuration file. We manually examined the extracted regular expressions to obtain a breakdown of the most targeted pages, so to understand where injections typically occur. We distinguish between *login pages*, *post-login pages*, *home pages*, and *all pages* (those regular expressions that cover all the pages of a given domain, such as **exampledomain.com/**). We were not able to classify the 19.5% of the regular expressions, because they did not contain any significant word in the path (e.g., login, logon, access), and the URLs were inactive. As shown in Figure 2.5, the majority (50.6%) of the sample we analyze targets all the pages of the domains. This strategy is more effective for cybercriminals, because their injections can still succeed even if the URL of the targeted page changes. Instead, regexes targeting post-login pages are a small fraction (9.5%). Thus, our current approach can be used to characterize the majority of the injection behaviors, leaving post-login pages as future work.

As a consequence, extracting these artifacts represents a further indication of the malicious activity, allows us to validate the results of the web page differential analysis and gain further insights about the use of WebInject techniques.

2.4 Application Scenarios

Prometheus produces signatures in the form of XPath expressions (Figure 2.6), which allow to check, on the client side, whether a web page is currently being rendered on an infected machine or, more in general, if a page of interest is targeted by a specific sample. In practice, as depicted in Figure 2.7, we foresee a centralized server and several consumers that send URLs of interest.

In **Scenario 1** the URLs are received by the clients (e.g., antivirus module, browser-monitoring component). The server replies with the list of signatures related to the requested URL(s). In the case of an antivirus, the browser-monitoring component (similar in spirit to Google Safebrowsing) can request the signatures of each browsed URL, and verify if any of the signatures match.

The second scenario that we envision, **Scenario 2**, has to do with research and large-scale monitoring. More precisely, we believe that Prometheus could be a good companion for initiatives such as ZeuS or SpyEye Tracker, VirusTotal, Anubis, Wepawet and similar web services that receive large daily feeds of malware samples and URLs to analyze. In this context, Prometheus can be used to automatically determine whether a sample performs web injection against a given URL, regardless of whether it is ZeuS or SpyEye, or some other unknown family, and to extract the portion of injected code.

One last application, **Scenario 3** envisions an IT administrator or web developer (e.g., of online banking backends), which provides a feed of URLs likely to be targeted by WebInject-based malware once in production. This scenario was suggested by a developer of a large national bank with which we collaborate, who noticed the lack of a centralized solution to determine whether their clients were infected by a banking Trojan. In this context, the developer would like to have a web framework that offers an API to programmatically mark sensitive resources (e.g., /page/login/, or those that contain forms). Marked resources will be processed by the web framework right before the HTTP response is sent to the requesting client. The web framework will then append a JavaScript procedure that, once executed on the client, performs a similar check to the one described in the aforementioned “Safebrowsing” scenario.

Generally, we envision our system to be deployed in collaboration with banks that provide feedbacks whenever their websites are massively updated, and strictly collaborate to update signatures. This would avoid almost all the false differences, and the possibilities for attackers to hide their injections. Moreover, as discussed in §4.6,

```
{
  "signatures": [
    {
      "xpath": "/html[1]/body[1]/table[3]/tr[1]/form[1]/input[13]",
      "value": "<input name=OTP type=password/>"
    }
  ]
}
```

Figure 2.6: Example of a generated signature for a given URL and sample

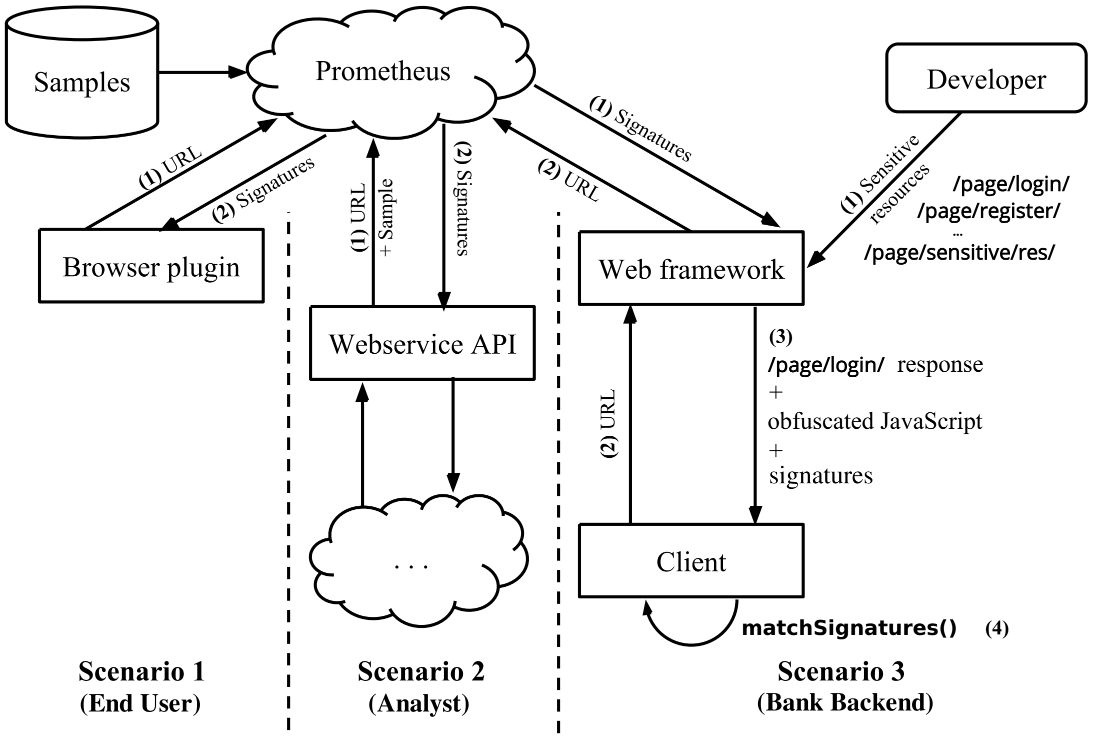


Figure 2.7: Three application scenarios described in §2.4

the signature-matching algorithm can be designed to match only the leaf nodes of the XPath, hence being more flexible with respect to legitimate page changes.

2.4.1 Detecting WebInjects through Live Memory Inspection

With Prometheus, we introduced an automatic framework for analyzing WebInject-based Trojans. Prometheus observes the differences that Trojans produce in an infected DOM and generates precise signatures of the injection behavior. Although our experiments showed the effectiveness of Prometheus for malware analysis, we did not fully exploit the generated signatures for *detecting* banking Trojans. Indeed, detecting widespread classes of malware by looking at their "generic" effects, instead of focusing on the specific implementation, already proved its efficacy in other contexts [55].

To overcome this limitation, we propose Iris [66], a client-side kernel-space module to automatically detect Man-in-the-Browser attacks that result in visible DOM modifications, independently from the malware implementation. Our system analyzes the memory of running browser processes and, leveraging live memory analysis, reconstructs the DOMs of the visited web pages to match injection signatures and spot artifacts of malicious web-injections.

However, signature matching in "raw" memory is not trivial and it requires to overcome several technical challenges. First, malware can easily circumvent any detection performed in the browser context—it infects and controls the browser. For this reason, we operate at a "lower level" (kernel driver). In addition, there is a semantic mismatch between the dynamic DOM objects allocated by the browser and the "raw" and static memory view that our solution must efficiently analyze.

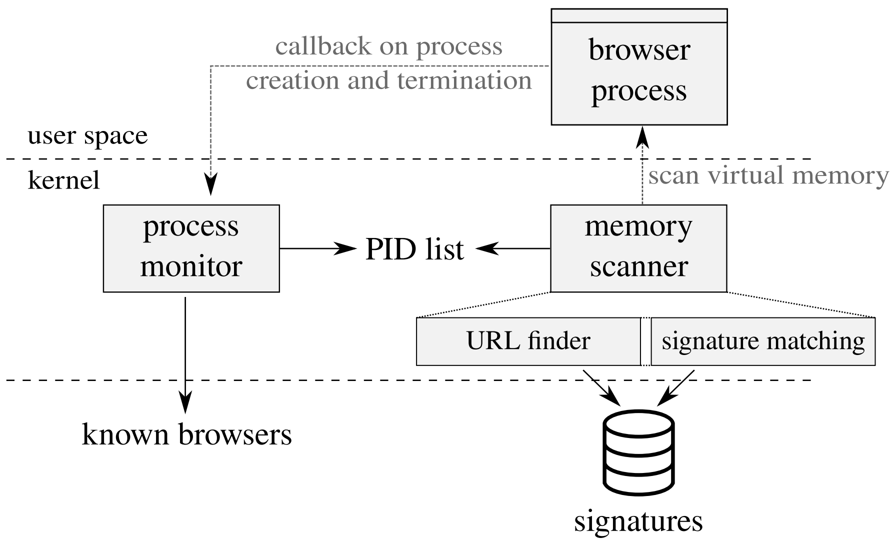


Figure 2.8: Architecture of Iris.

We implement a prototype of Iris as a Microsoft Windows kernel driver, and we evaluate it on two distinct ZeuS and Citadel samples, analyzing a real, live banking website. We manually verified these preliminary results, showing that Iris is able to detect when a browser is infected.

Iris is composed of two parallel processes (Figure 2.8): a process matcher and a memory scanner. The *process matcher* holds a list of PIDs belonging to running browser processes: It is notified whenever a new process is created or deleted, and, if the process image name belongs to a known browser, updates the PID list. Periodically, the *memory scanner* scans the memory space of each browser process, extracting the URLs of the pages being visited. If the signature database contains signatures related to the extracted URLs, it invokes the signature matching module, which scans the process memory to match its content (i.e., in-memory DOM fragments) with known signatures. If there is a match, we detect an infection.

Signatures. A signature is a tuple containing (a) a URL, (b) an XPath expression specifying the location of the injection in the DOM, and (c) the injected content, which can be a node or an attribute. Such signatures are URL-specific and are based upon the *effect* of the banking Trojan on the target web-page rather than the malware implementation: They can be generated through automated dynamic analysis, such as Prometheus.

Memory Scanner. The memory scanner cyclically iterates through the running browser processes' PIDs and gathers the valid (i.e., allocated) virtual memory pages. It scans the process memory space to determine the memory state (`ZwQueryVirtualMemory`). If a page region has state `MEM_COMMIT`, it is valid and can be scanned for URLs or signatures: Through the `KeStackAttackProcess` routine, Iris attaches to the target process memory space and makes a copy of the memory region to analyze. The copied memory buffer is passed to the URL finder and signature matching modules. After that, we move to the next region of addresses in the target process, repeating this procedure until a region has an invalid state (`STATUS_INVALID_PARAMETER`).

URL finder. This module finds the visited URLs in a memory region belonging to a running browser process. In general (e.g., Internet Explorer), multiple browser tabs share the same process: URLs and DOMs are scattered over the process memory. Hence, we need to perform a preliminary complete scan of the process memory to gather the relevant URLs. Instead, the multi-process architecture of some browsers, such as Google Chrome, has a different rendering process for each tab (i.e., each visited URL). In this case, we stop the linear memory scan after identifying the visited URL, performing signature matching only over the remainder of the process memory. This is possible because, as experimentally determined, in the memory address space of each Chrome rendering process, the URL is in a lower memory address than the DOM.

Signature matching. This module matches a memory buffer against the signatures targeting the visited URLs in three steps:

1. *String search.* First of all, it searches for occurrences of the signature content (i.e., the injected HTML/JS code) in the memory buffer using the Boyer–Moore algorithm [17]. This allows to efficiently discard signatures that do not match.

2. *Refinement.* If there are candidate matches, it refines the search by checking whether the string found in memory is part of a valid HTML fragment (e.g., in case of an attribute injection, the value found in memory should be an attribute with the same HTML element type described in its signature). We search the start tag of the element backward in memory, starting from the matched value; when we find the node, we use a lightweight HTML parser to validate that the string ranging from the start tag until the injected value is a valid HTML element with the type and attribute (or content in case of text injection) defined by the signature. This phase is not required when whole HTML fragments are injected.

3. *XPath search.* Lastly, it checks whether the *location* of the match in the DOM is the one specified in the signature's XPath expression. We proceed by walking the XPath expression starting from the node with the injected content and moving backward to the DOM root, while matching DOM nodes in the memory dump. We note that, often, the complete DOM continuously changes, and cannot be retrieved with a single live memory "snapshot." Instead of attempting to reconstruct the DOM correlating multiple memory scans, we overcome this issue by coping with partial and approximate matches (e.g., when only fragments of the DOM are available, or when the DOM is scattered around the memory).

Iris supports three levels of match: In the best case, it matches the XPath expression in the signature up to the root node; on average, it partially matches the XPath expression; in the worst case, it matches only the signature value. The user can tune the match level required to trigger a detection according to the desired trade-off.

Preliminary Results In order to test the correctness of our tool, we performed an experiment building a custom Trojan in a controlled environment. Specifically, we got access to ZeuS and Citadel's builders. Using such tools, we built two samples defining a list of custom web-injections for a real banking website. We then manually created the signatures for such injections and run the samples in a controlled environment (VirtualBox VM running Windows 7 32 bit) where we previously installed Iris (together with our signatures).

As a result, IRIS detected the presence of the Trojans when we visited the targeted web-pages using Internet Explorer, showing it is able to successfully match such signatures only looking at the raw memory of the browser.

Future Work. We plan to extend the evaluation of IRIS on a large dataset of distinct samples of Trojans. In particular, we want to study how the different matching techniques (full XPath, partial XPath, content only) affect the performance of the detection.

2.5 System Design & Implementation

The input of our system is a list of URLs that need to be monitored (e.g., banks URLs) and a malware sample. Given the inputs, the system performs the web page differential and the forensic memory analysis in three phases: (1) Data Collection, (2) Data Processing, and (3) Signatures Generation.

As a final output, our approach produces a list of differences per URL, which precisely identify the portion of injected or changed code and represent our signatures. Each signature is defined by (1) the XPath of the affected node or attribute, and (2) the injected content (Figure 2.6). As a matter of fact, these signatures partially reconstruct the configuration file.

In Figure 2.9 we reported the overview of our analysis, given a malware sample to analyze. In the following sections we provide the details of each phase.

2.5.1 Phase 1: Data Collection

Our system retrieves (1) the DOMs objects, and (2) the memory dump of the web browser process. To this end, the system executes a small set of virtual machines. Half of them are infected with the malware sample and the other half are clean. Afterwards, the system visits the set of URLs with each VMs, collects the DOMs and the memory dump.

DOMs Collection. Each VM receives a list of URLs as an input, and it visits each URL with a `WebDriver`-instrumented browser. For each visited URLs, PROMETHEUS saves the resulting DOMs and it stores them as serialized representations of existing nodes, including any DOM manipulations performed by client-side code at runtime while the page loads. The DOMs comprise the content of the nodes in the pages, including script tags. Since the content of a web page may vary legitimately (e.g., server-side scripts, advertisement inclusions), the DOMs collection phase is performed on multiple clean and infected machines. This is used to eliminate legitimate differences that occur between two or more clean VMs. When this phase terminates, all the dumped DOMs are stored and labeled as "clean" or "infected."

Memory Dump In this phase, PROMETHEUS executes a new VM along with the target malware sample. To trigger the malware infection and obtain the target URLs, PROMETHEUS runs the browser inside the VM. Then, the system dumps the memory of the VM from the host OS by using the `Cuckoo` feature. This allows us to avoid creating any artifacts that can be exploited by the malware. At the end of this phase, the memory dump is stored as a file on the disk. We use a separate VM for the memory analysis to avoid false positives due to the execution of multiple applications on the same VM.

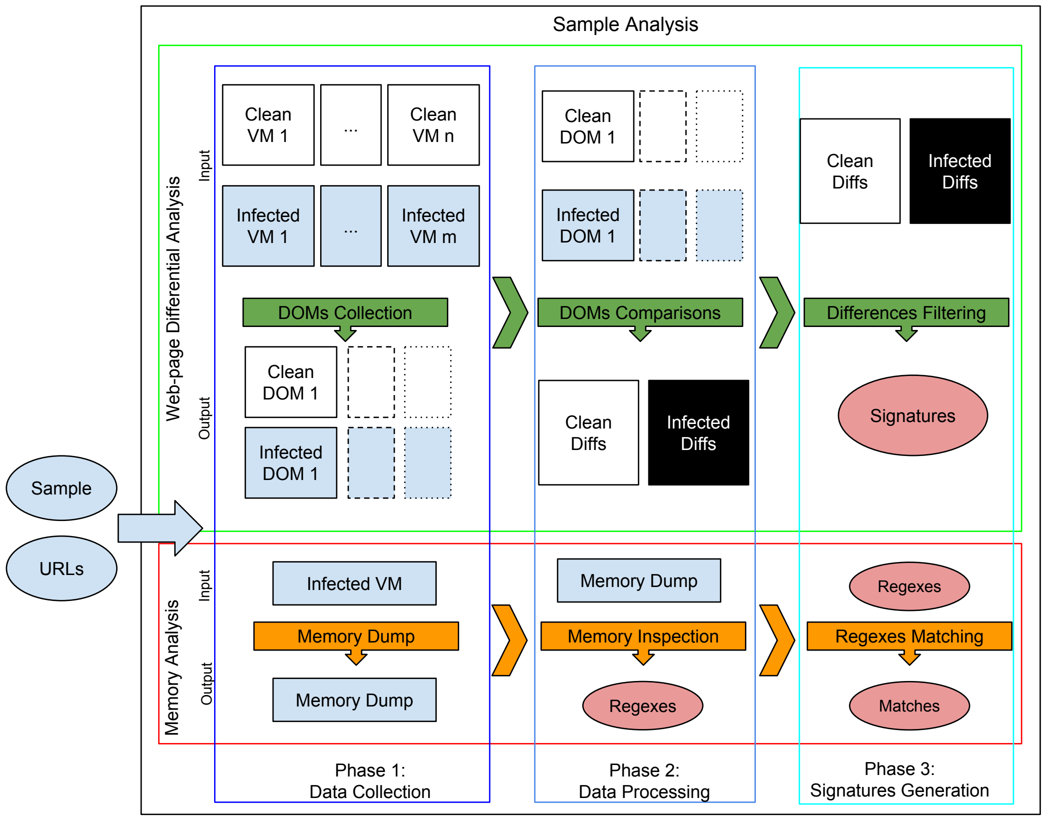


Figure 2.9: Overview of a sample analysis. The web page differential analysis, and the memory analysis are performed in parallel.

2.5.2 Phase 2: Data Processing

During the second phase of our approach, Prometheus compares the various DOMs retrieved for each URL (downloaded by distinct VMs) and extracts the WebInject target URLs from the memory dump obtained in the previous phase.

DOMs Comparison. We compare the DOMs in the following way. For each URL, we consider one random clean DOM as a reference, and we compare all the others against it. We rely on `XMLUnit`'s `DetailedDiff.getAllDifferences()`, which walks the DOM tree, and extracts the following differences:

- **Node insertion**: Most information stealers add new fields in forms, injecting one or more `<input/>` nodes. Thus, it is crucial to identify this case, as it is one of the most common injections.
- **Attribute insertion**: This identifies injections that are mostly related to malicious JavaScript code. In the common case, Trojans add attributes such as `onclick` to bind JavaScript code and perform malicious actions whenever certain user-interface events occur.
- **Node modification**: This occurs when Trojans modify the content (e.g., text enclosed in) of an existing node. Often the target node is `<script>`.

- **Attribute modification**: This occurs when Trojans change the value of an existing attribute (e.g., to change the network address of the server that receives the data submitted with a form, or to modify the JavaScript code already associated to an action).

At the end of this phase, for each processed URL the extracted differences are inserted into two lists (clean differences and infected ones), according to the compared DOM label. Each difference is composed by three elements:

- **Type**: The nature of the difference (node insertion, node modification, attribute insertion, or attribute modification).
- **XPath**: The XML path to the node that is affected by the difference.
- **Content**: The value of the difference (e.g., in the case of a node insertion the content is the HTML node inserted with all its own attributes).

Memory inspection. When the infected memory dump is generated, it is inspected in order to extract the WebInject targets. To extract the target URLs and regular expressions, we developed a `Volatility`[3] plugin based on `YARA`[4]. This plugin scans the memory dump, looking for all the strings that match a `YARA` rule. In particular, since we observed that the URLs and the regular expressions are loaded in the browser's memory, the plugin inspects only its address space. We defined a regular expression that matches the pseudo-URL format of the WebInject target URLs (e.g., `domain.com/ibank/transfers/*`, `bank.com/login.php*`). Moreover, our `YARA` rule filters the matched strings that are close to each other. In fact, since we noticed that the WebInject targets are allocated sequentially, we leverage this fact to exclude all the matching strings that are not WebInject targets.

2.5.3 Phase 3: Signatures Generation

Prometheus now filters out the legitimate differences employing four heuristics. Moreover, it exploits the information extracted by memory analysis in order to validate the generated signatures.

Differences Filtering. The two lists of differences produced in the previous phase are filtered according to four heuristics. The objective is to eliminate the legitimate differences, therefore reducing the false difference rate. We designed and implemented the following four heuristic filters:

Heuristic 1: Whitelisting Clean Differences. In certain pages, there may be some nodes that change very often their content (e.g., calendar, clock, advertisement and so on). This kind of nodes generates a lot of differences that refer to the same node but with different content. These differences are present in both the clean and infected list. For this reason, we remove all the infected differences with the same type and the same XPath of a clean difference. For example, thanks to this filter we are able to discard the differences caused by advertisements that dynamically change their message.

[3] `https://github.com/volatilityfoundation/volatility/` [4] `http://plusvic.github.io/yara/`

Heuristic 2: Ignoring Node or Attribute Reordering. In other pages, it may happen that some nodes with a fixed content (mostly JavaScript) are omitted or moved in different places inside the web pages. We remove all the infected differences that have the same type, the same last node in the XPath, and the same content of a clean difference. For example, this filter discards those differences caused by an advertisement that presents a fixed content but that is dynamically loaded in different positions of the page. Even if this scenario seems unlikely, we found it occurring in different web pages.

Heuristic 3: Filtering Harmless Differences. Since malware authors are inserting new elements in order to steal data from the victims, we are interested in differences that are caused by insertions or modifications. For this reason, all the differences that indicate other changes (e.g., deletion of nodes) are filtered out. Moreover, we whitelist attributes that are harmless according to our attacker model (i.e., value, width, height, sizset, title, alt) and PROMETHEUS automatically filters them out.

Heuristic 4: Identifying Malicious Injections. A typical injection has the following characteristic: it is present in all the DOMs downloaded by infected machines, but not on the DOMs downloaded by clean machines. Furthermore, an injection is defined to insert a content after, or before, a node. Thus, we can assume that, for a give sample and URL processed by multiple VMs, an injection has always the same content, and injects in the same node, even if the node changes its XPath. So a typical web injection refers to the same last node of an XPath that may vary somehow. Hence, we filter out all the differences that are not present (with the same type, the same last node of the XPath, and the same content) in at least $\varepsilon\%$ of the infected DOMs. Since a sample may not activate on some VMs (e.g., because malware authors take countermeasures to prevent dynamic analysis) it may happen that a sample manifests its behavior only on a subset of the VMs. The threshold ε copes with this problem.

The infected differences that pass this filtering process are considered as malicious.

Regular Expression Matching. In this last phase, the regular expressions extracted through memory inspection are examined to validate the results of the web page differential analysis. If an injection is found at a URL not matching any of the regular expressions, it is probably a false difference. Second, the number of matching regexes allows us to rank the URLs according to how often they are targeted. This is useful in case we want to limit the number of URLs processed during an analysis.

2.5.4 Implementation

The PROMETHEUS prototype that we implemented is composed by a back-end and a front-end. The back-end receives as input the specification of the submitted analysis, then it schedules it, managing the available resources (VMs), it processes the data as explained in §2.5, and stores the results. We integrated PROMETHEUS with `Cuckoo`[5], an open-source sandbox that interacts with the most common virtual machine managers. We used `Oracle VirtualBox`[6] as VMM. Inside each VM, we installed and configured

[5] `http://www.cuckoosandbox.org/` [6] `https://www.virtualbox.org/`

`WebDriver`[7], a platform- and language-neutral interface that introspects into, and controls the behavior of, a web browser and dumps the DOM once a page is fully loaded.

The front-end is a web application, through which users can submit samples, or URLs, and obtain the results of their analysis or previous ones. Finally, we implemented a JavaScript web-socket in order to dynamically show results during the analysis.

2.6 Experimental Results

We evaluated our implementation of Prometheus against 68 real, live URLs of banking websites targeted by 135 distinct samples of information-stealing malware. We manually verified the activity of each sample when the browser was rendering the targeted websites, to make sure that all the samples were working as expected. While doing this, we conducted a preliminary experiment to measure the dormant period after which the malware triggers its malicious injections (these results are presented in §2.6.2). We found out that 33.82% of the web pages were affected by at least one injection, which Prometheus detected correctly, as summarized in Table 2.1.

Our first goal was to measure the correctness of the signatures that Prometheus generates (§2.6.3 and 2.6.4), and the outcome of the memory analysis (§2.6.5). Then, we assessed the computational resources required by Prometheus in function of the number of DOMs compared per URL (§2.6.6). Last, we evaluated the impact of false positives caused by our signatures (§2.6.7) during a typical real user's browsing activity.

In all our experiments we deployed Prometheus on a 2.0GHz, 8-core Intel machine with 24GB of memory. We used VirtualBox as a virtual machine monitor, and each VM was equipped with 1GB of memory, enough to run Internet Explorer 8 on top of Windows XP SP3.

2.6.1 Dataset

Our dataset is based on the ZeuS family, which is by far the most widespread information stealer that performs injections. According to the conservative statistics by ZeuS Tracker, as of February 23, 2016 there are 518 known C&C servers (179 of which active), and a low estimated antivirus detection rate (40.05%, zero for the most popular and recent samples). We also conducted a series of pilot experiments with SpyEye, which is less monitored than ZeuS; thus, it is more difficult to obtain a large set of recent samples. However, SpyEye uses the same injection module of ZeuS, as described by Binsalleeh et al. [14], Buescher et al. [18], Sood et al. [83]. For these reasons, for the purpose of evaluating the quality of our signature-generation approach, we decided to select ZeuS as the most representative information stealer that generates real-world injections. We tested 196 (and counting) samples, but 61 of these failed to install or crashed, leaving 135 distinct samples.

We constructed a list of target URLs starting from a webinjects.txt leaked as part of the ZeuS 2.0.8.9 source code[8], and adding some new URLs extracted from initial memory analyses. However, we removed most of the URLs contained in the initial list, because they were inactive. The final URL list that has been evaluated was composed

[7] `http://docs.seleniumhq.org/projects/webdriver/` [8] `https://bitbucket.org/davaeron/zeus/`

by 68 distinct URLs. Building a list of URLs from webinjects.txt files found in the wild allowed us to deal with real-world targeted pages.

Finding a way to measure the quality of the evaluation results was not an easy task, since to determine the real injections performed by a sample we should have decrypted and checked its own WebInject configuration file. We initially created a controlled botnet, and we built a ZeuS sample with a customized webinjects.txt. However, we wanted to test Prometheus on real samples found in the wild. Therefore, we decided to manually analyze the results provided by Prometheus to assess their correctness. We exploited the knowledge created by memory forensic analysis in order to validate the results. In details whether a difference is detected on a URL that does not match any of the regexes extracted from infected memory dumps, the system flags it as a false difference. This reduced the volume of data to analyze manually without introducing any experimental biases.

2.6.2 Delayed activation

One common anti-analysis technique is to delay the real execution of a malware sample. To this end in order to conduct a sound analysis, it is necessary to check the time in which a sample is activated. For this reason, we conducted some experiments in order to estimate the activation time to be used. In the further experiments, all the infected VMs wait for such time slot before starting the analysis.

In order to estimate the activation time we implemented a specific analysis package for `Cuckoo`. The module works as follows. `Cuckoo` starts a virtual machine and executes the selected malware sample. Afterwards, it starts the browser and waits for a prefix amount of time. When the time is expired, the system dumps the application memory, and terminates the VM. In the next step, we examined the memory dump with Volatility. In particular, we used the `apihooks` [9] plugin to search all the API hooks installed by the malware. We are interested in looking for any hook on the `WININET` DLL. In fact, these hooks are installed by malware to perform Man in the Browser attacks. Hence, when we find any of these hooks it means that the sample was active. Then, to estimate the activation time we repeated our analyses applying a bisection algorithm in order to find, with a reasonable precision, the instant of activation of the sample. We repeated these experiments on different active samples. We obtained several empirical results and in the worst case the activation time was 50 seconds.

2.6.3 False Differences discussion

A false difference occurs when Prometheus detects a benign difference in a web page and classifies it as a malicious injection. We observed that the main cause of false differences are JavaScript-based modifications. Most of the modern websites contain JavaScript code that modifies the DOM of the web page at runtime, loading dynamically changing content. Furthermore, sometimes server- side scripts generate different JavaScript code every time the page is requested (e.g., advertisement). Even so, we managed to discard almost all the legitimate differences and, with all our heuristics enabled, the overall false difference rate is 0.70%. Furthermore, most of false differ-

[9] `https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/apihooks.py`

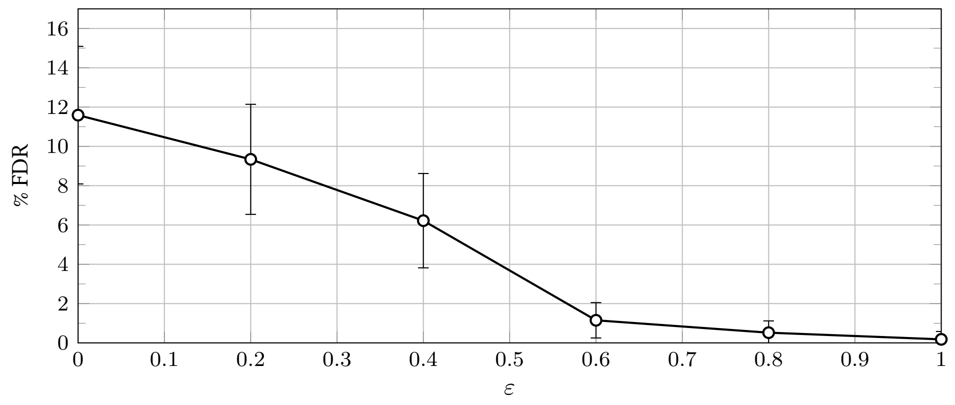


Figure 2.10: False Difference Rate depending on the ε threshold evaluating 68 distinct URLs and using 12 VMs (6 clean ones and 6 infected ones).

ences are distinguishable looking at the results extracted by memory analysis, since a difference detected on a URL that do not match any of the extracted regular expressions is certainly a false difference. Therefore, validation through memory forensic reduces the false difference rate to 0.26%. In particular, we preferred to not automatically discard the differences related to URLs that do not have references in memory. This is done because, in the case the memory analysis fails in extracting some regular expressions, we still consider all the malicious differences. We analyzed the effects of the number of VMs, and of the threshold ε (**Heuristic 4**) on the false differences (Figures 2.10 and 2.11). We observed that the number of VMs is the most effective parameter in the reduction of false differences. However, since the number of VMs required to guarantee a low false difference rate is high, at least 25, and this can worsen the performance because of the overloading, using a high value of ε helps in guaranteeing such results with a lower number of VMs (10-12).

One of the previous heuristics used by Zarathustra ignored dynamic DOM differences. The assumption was that malware always inserts at least one static node or attribute, which would be still visible even when JavaScript is disabled. This heuristic had proved to be the most effective in reducing false differences. However, we found samples that perform only JavaScript injections, therefore we disabled such assumption. Figure 2.12 shows Zarathustra's false difference rate when this heuristic is disabled. While Zarathustra reaches 1.0% of false difference rate, PROMETHEUS reduces the false difference rate even using less VMs (Figure 2.11).

Table 2.1: Most injected domains

Domain	Avg # Injections
`ybonline.co.uk`	10.244
`cbonline.co.uk`	9.723
`lloydstsb.com`	7.482
`bbvanetoffice.com`	4.275
`banesto.es`	1.620
`gruppocarige.it`	1.121
`scrigno.popso.it`	0.916
`isideonline.it`	0.861
`wellsfargo.com`	0.861
`uno-e.com`	0.747

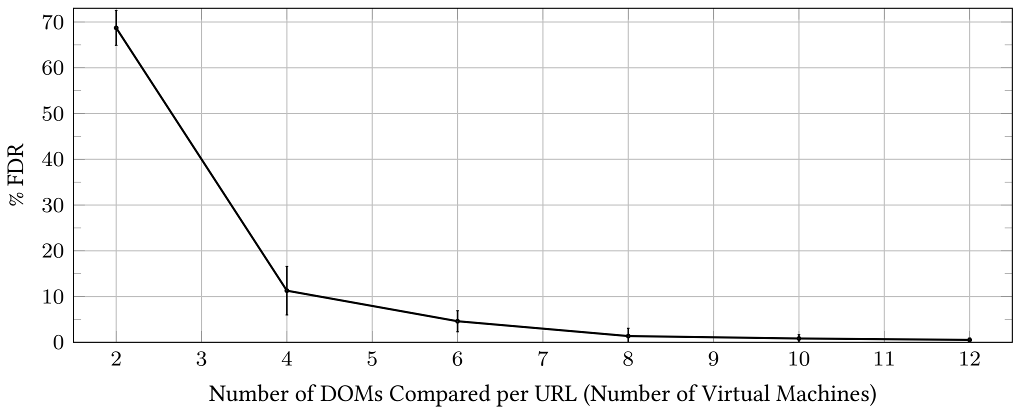


Figure 2.11: False Difference Rate depending on the number of VMs used processing 68 distinct URLs with threshold $\varepsilon = 0.8$.

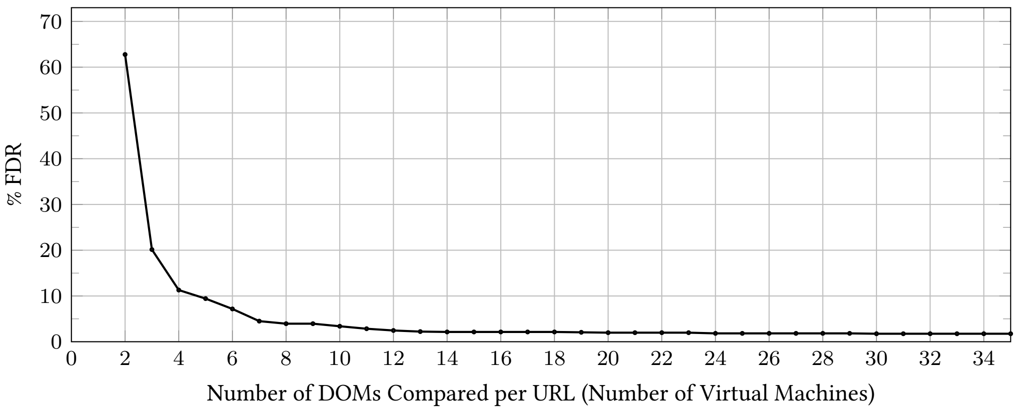


Figure 2.12: Zarathustra's false differences for an increasing number of clean VMs.

2.6.4 Missed Differences discussion

A missed difference occurs when PROMETHEUS does not identify a malicious injection in a web page. Since the DOMs comparison process is deterministic, there are only two cases in which this can happen. The first one occurs when a high threshold ε is used in **Heuristic 4** and a sample fails to execute in some of the infected machines. As explained in §2.5.3, the differences that are not present in most of (depending on ε) the infected machines are filtered out, so if a sample manifests its behavior just in few of the infected machines, for example if it uses a randomized activation time, its injections might be discarded. Although we did not observe such behavior, this problem can be easily solved by properly tuning the sleep time and the ε threshold. The second case is also related to **Heuristic 4**. Since the heuristic is based on the assumption that each sample injects a static content, if a sample injected different dynamic content on the same web page every time it is executed, it would evade our system. However, this case never happened during the evaluation, and all the samples in our dataset performed static content injections. Samples injecting dynamic content could require visiting the same pages multiple times on each VM.

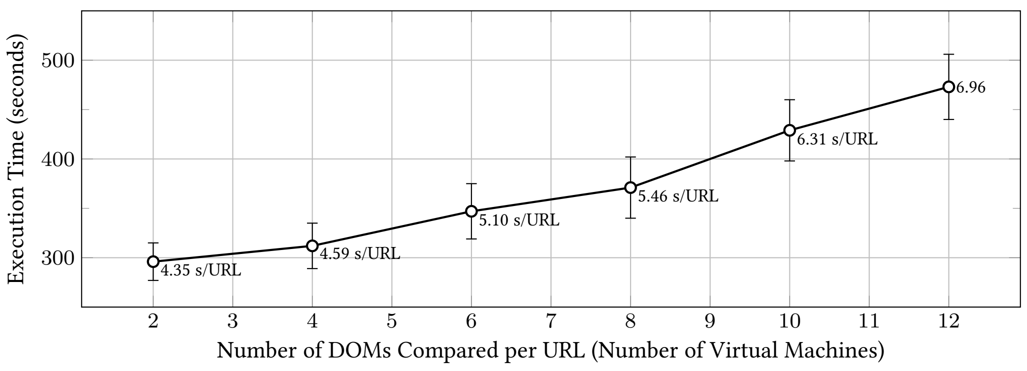


Figure 2.13: Speed and Scalability of Prometheus: Mean time required to process 68 URLs for each sample. The labeled points indicate the mean time required to process a single URL.

2.6.5 Results of memory analysis

Prometheus extracted 694 distinct regular expressions through memory forensics (Table 2.2). As we expected, we observed that true injections were present in URLs that matched some of the extracted regular expressions. This is a further proof that can help to distinguish real injections from false differences. Another important finding that emerged from the results is that not all the regular expressions imply injections. There are some cases in which some URLs were not injected, even if they were present in the memory-extracted targets list. The cause of this could be that the injections failed because the hooking points configured in the WebInject configuration file were wrong or old (because the web page changed and does not contain that HTML code anymore), or simply the sample just monitors the URLs stealing the data submitted by the victims, without injecting new contents.

In conclusion, our results show that combining both the web page differential analysis and the memory forensic inspection is important in order to gain more insights on the WebInjects' behavior, and to guarantee low false differences.

2.6.6 Performance

We measured the execution time of Prometheus. Prometheus has been designed and implemented to parallelize all computations. Furthermore, the approach used to perform DOMs comparisons is asynchronous, and all computations are executed in parallel as soon as the required data is available. For this reason, the execution time is dominated by the time required by the VMs to sequentially visit each URL and dump its DOM, while the time required to compare DOMs and generate the signatures is negligible. Prometheus performs a sample analysis on 68 URLs using 10 VMs in about 7 minutes. The sample analysis includes also the memory forensic inspection, which

Table 2.2: Top five regular expressions extracted by the memory analysis.

Regular Expression	Frequency
`http://*odnoklassniki.ru/*`	81.3%
`http://vkontakte.ru/*`	81.3%
`*odnoklassniki.ru/*`	81.3%
`https://online.wellsfargo.com/das/cgi-bin/session.cgi*`	76.4%
`https://ibank.barclays.co.uk/olb/x/LoginMember.do`	75.6%

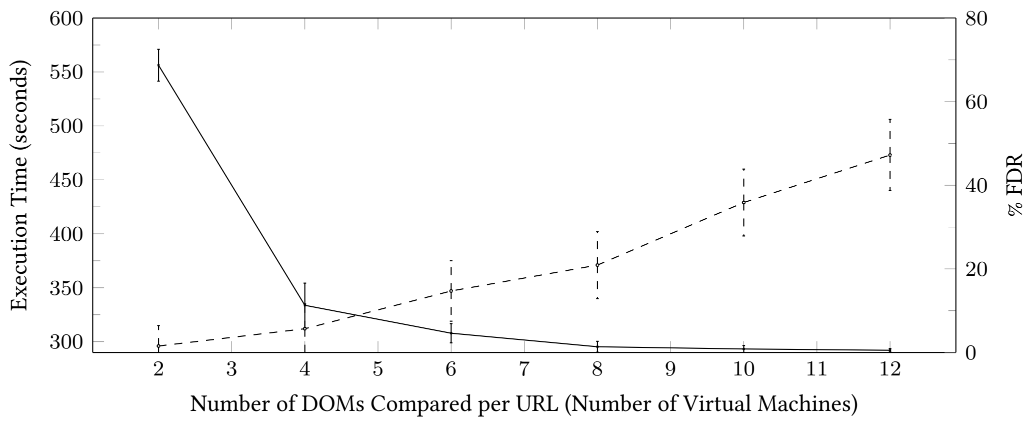


Figure 2.14: Trade-off between Performance and False Difference Rate. The dashed line refers to the execution time required to process 68 URLs. The solid line refers to the False Difference Rate analyzing 68 distinct URLs with threshold $\varepsilon = 0.8$.

is performed in parallel and requires less time than the web page differential analysis. As shown in Figure 2.13, Prometheus is able to process each URL in little more than 4 seconds when 2 VMs are used. The chart shows that Prometheus scales well increasing the number of VMs, with just a little overhead. However, when the number of VMs is higher than 10 the overhead slightly increases. This is due to the overload on the single physical machine, and to the fact that all the VMs network traffic flows through the single virtual interface between VirtualBox and the host OS. Figure 2.14 shows the trade-off between performance and false difference rate depending on the number of VMs.

We measured also the amount of memory required by Prometheus. Prometheus required at most 15 GB of RAM for a sample analysis using 13 VMs, each of them set with 1 GB of memory. However, since VirtualBox allocates the entire amount of memory assigned to the VMs, we inspected the VMs from their internal, and we measured that each VM required about 300 MB.

2.6.7 Distributed crawling experiment

We ran a distributed crawling experiment to evaluate the generality of the signatures generated by Prometheus. With generality we mean the applicability, with an acceptable false positive rate, of a signature to an arbitrary URL. We developed a Chrome extension that sends to our secure servers the DOMs of the visited web pages. Between October and December 2014, we distributed our Chrome extension to trusted users and we collected 8,226 DOMs. To guarantee users' privacy, we did not record any information about them, keeping data anonymized. We further set up a machine on which we installed and configured `WebDriver` to control a Chrome browser, with our extension embedded, and we automatically downloaded the DOMs of the home pages of the top 10,000 Alexa[10] websites. We then checked the signatures generated by Prometheus on the 18,226 DOMs collected, assuming that such DOMs are clean.

As we said our signatures are formed by an XPath and a value. Furthermore, each of our signatures refers, by design, to the URL from which it was generated. That means that each signature can be used to check, client-side, if the web page of the specific

[10] `http://www.alexa.com/topsites`

URL is currently being rendered on an infected machine. Instead in this experiment, we want to ignore the URLs from which the signatures are generated, and measure the amount of DOMs matching at least a signature. In this way, we can evaluate the generality of such signatures, and measure their dependency from the URLs they refer to.

We performed the matching process twice. In the first case, we considered only the XPath to verify the match of a signature against a DOM. In the second case, we considered both the XPath and the content of the signatures. We made also another distinction, checking all the generated signatures, or only those validated by the memory analyses. Looking just at the XPath, the percentage of DOMs matching at least one signature is very high: 97.74% considering all the signatures, and 94.43% considering those validated through memory forensic. This is mainly caused by signatures affecting *script* tags (e.g., */html[1]/body[1]/script[2]*) that are injected by cybercriminals to verify, client-side, the presence of all the fields in the forms. Checking both the XPath and the content of the signatures, the percentage reduces but it is still quite high: 25.99% considering all the signatures, 24.86% only those validated through memory forensic. This is caused, for example, by signatures relating the import of `JQuery` (e.g., *"xpath": "/html[1]/body[1]/script[2]", "value": "@src='https://ajax.googleapis.com/ajax/libs/jqueryui/1.7.1/jquery-ui.min.js' "*), used by the attackers to create and inject fancy forms. Since the amount of DOMs matching at least a signature is high in all the cases, we cannot ignore the dependency of the signatures from the URLs they refer to, as doing so will cause a high false positive rate during the signature matching process.

In conclusion, this experiment demonstrates that the signatures are strictly dependent on the URLs from which they are generated, and shows the necessity to design a more sophisticated signatures-matching algorithm that takes in consideration the relation between signatures and URLs.

2.7 Limitations

One limitation of PROMETHEUS is the assumption behind **Heuristic 4**: The content of the injections performed by WebInject-based information stealers is assumed to be static, which implies that the injections performed by the same malware have always the same content. The assumption held for all the samples we analyzed, but Boutin [16] observed a new WebInject mechanism, in which the injected content is delivered by the C&C server each time the injection is performed. Performing content-dynamic injections, this mechanism could evade our system. Therefore, our approach will need to be revised, refining the set of heuristics, or visiting the same page multiple times on each VM in order to extract all the variants.

Our second discussion point is that malware operators could rewrite the injected code, introducing no-op DOM nodes with the goal of evading the signatures generated by PROMETHEUS: adding an additional `<div/>` wrapper to a page (in a random position), for instance, would circumvent a naive use of our signatures (i.e., if the full XPath is considered from the root to the leaves). However, none of the samples in our dataset adopted this technique. In addition, although we leave the implementation of a proper signature matching algorithm to future work, we are aware that there is an accuracy trade off between matching the entire XPath expression of a signatures versus matching only the leaf nodes. In the latter case, the signature verification process will be more flexible and can be effective even in the case of pages subject to large modifications.

Third, the memory forensic analysis could be evaded by samples using custom sparse data structures, for example splitting the regexes and storing each part separately. In this case the analysis would require more expensive carving techniques to be able to identify and extract the WebInject targets.

Since we take the original banking website as an oracle, an injection that matches exactly with a benign difference would not be considered as malicious. For example, this happens if the website is updated with a new form input that matches the very same XPath expression of an injection. Not only this is very unlikely to happen, it is also very easy to remediate by leveraging feedback from the bank whenever its site is updated, or possibly by requesting an update of the signatures for that domain. It is indeed reasonable to envision PROMETHEUS deployed within a bank information system: this use case would avoid most, if not all, the venues for false differences as a fully up-to-date model of the clean website would always be available. Similarly, PROMETHEUS can easily monitor authenticated web pages, which are not a limitation when our system is deployed by the website provider (e.g., bank).

Finally, another obstacle is malware that adopts anti-analysis techniques. However, PROMETHEUS can be ported, as is, on a bare metal hypervisor [52]. Even if certainly not impossible, it is definitely harder to detect bare metal environments.

2.8 Concluding Remarks

In this chapter we presented PROMETHEUS, an automated system to analyze the client-side behavior of financial Trojans that perform web injections. PROMETHEUS generates signatures comparing the different DOMs retrieved by infected and not-infected VMs. These differences are then filtered by using some heuristics, in order to discard legitimate ones. This approach is combined with a memory forensic inspection, to extract the WebInject target URLs and validate the signatures generated by the web page differential analysis. The main advantage of this approach is the independence from the implementation details of the analyzed malware. We evaluated PROMETHEUS on a dataset of 135 distinct samples of ZeuS, analyzing 68 real, live URLs of banking websites. The results show that PROMETHEUS correctly identified the injections performed by the analyzed Trojans with a low false difference rate (0.70%). Validation through memory forensic reduces errors down to 0.26%. Furthermore, PROMETHEUS scales well with the amount of available resources, and it is able to generate the signatures for 1 URL in about 6 seconds.

Besides the development of a proper signature matching algorithm, future research should concentrate on more advanced uses of WebInjects. As mentioned in the §4.6, dynamic-content injections (described in [16]) require visiting the same pages multiple times on each VM to capture all the different injections. Other advanced WebInjects, described by Kharouni [50], perform attacks that may not result in DOM modifications. An example is a banking web application that allows to divert a wire transfer by simply modifying one, single parameter in an outgoing HTTP request, the respective HTTP response (e.g., page that confirms the result of a transaction), and all the subsequent pages. When such fraudulent transfers to an attacker-controlled account have been made, these modules are able to hide the transactions and revise the current account balance in order to make the victim unaware of the fraud.

Finally, in this chapter we showed that the DOM is a simple yet effective observation point. We believe that other aspects of the browser behavior can be observed and compared on infected vs. clean clients, to assess whether the information stealers cause side effects in the browser that can be used as a detection criteria.

3. Protecting from Ransomware Attacks

Ransomware [104] is a class of malware that encrypts valuable files found on the victim's machine and asks for a ransom to release the decryption key(s) needed to recover the plaintext files. The requested ransom payment is typically in the order of a few hundreds US dollars [78] (or equivalent in crypto or otherwise untraceable currency [84]). Clearly, the success of these attacks depends on whether most of the victims agree to pay (e.g., because of the fear of losing their data). Unfortunately, according to a thorough survey dated November 2015 [10], about 50 percent of ransomware victims had surrendered to the extortion scheme, resulting in million of dollars of illicit revenue. In March 2014, Symantec estimated that the Cryptowall gang has earned more than $34,000 in its first month of activity. In June 2015, the FBI's Internet Crime Complaint Center [33] reportedly received 992 Cryptowall-related complaints between April 2014 and June 2015, totaling $18M worth of losses. In the first three months of 2016, according to a recent analysis [64], more than $209 million in ransomware payments were made in the US alone. From a technical viewpoint, ransomware families are now quite advanced. While first-generation ransomware were cryptographically weak, the recent families encrypt each file with a unique symmetric key protected by public-key cryptography. Consequently, the chances of a successfully recovery (without paying the ransom) have drastically decreased [7, 53].

Problem Statement and Vision. Kharraz et al. [51] were the first to analyze a large corpus of ransomware samples. The authors suggest that the filesystem is a strategic point for monitoring the typical ransomware activity. In this dissertation, we set the next research objective: Creating a forward-looking filesystem that transparently prevents the effects of ransomware attacks on the data. We make a step toward such vision by proposing, implementing and evaluating an approach that combines automatic detection and transparent file-recovery capabilities at the filesystem level, all combined in a ready-to-use Windows driver.

Preliminary Feasibility Assessment. Our first goal is to understand how ransomware *compares* to benign software from the filesystem's viewpoint. We start by analyzing in-depth how benign software typically interacts with the filesystem on *real-world* computers. We use the I/O request packets (IRPs) as the focal point of our analysis, as IRPs are the basic data units originating from high-level operations (e.g., read file, open file). In practice, we performed the first large-scale data collection of IRPs from real-world, ransomware-free machines, to profile the low-level filesystem activity in normal conditions. To this end, we developed IRPLogger, a data-collection agent that we installed on 11 machines used by volunteers for their typical day-to-day tasks (i.e., personal, office, and development). We anonymized and collected about a month worth of data, gathering more than 1.7 billion IRPs generated by 2,245 distinct applications (we will made this data available to other researchers). Using this collected data as a reference, we populated a set of analysis machines with files and directory trees such that they resemble the typical filesystem organization and content observed in the 11 real-world machines. This step is essential to create a realistic environment such that to trigger the ransomware attacks. We then used IRPLogger to monitor the filesystem on such machines infected by state of the art ransomware samples.

Proposed Approach. Our preliminary assessment guided us to design a detection system based on the combined analysis of entropy of write operations, frequency of read, write, and folder-listing operations, dispersion of per-file writes, fraction of files renamed, and the file-type usage statistics. Our approach is to automatically create detection models that distinguish ransomware from benign processes at runtime ShieldFS adapts these models to the filesystem usage habits observed on the protected system. Additionally, ShieldFS looks for indicators of the use of cryptographic primitives. In particular, ShieldFS scans the memory of any process considered as "potentially malicious," searching for traces of the typical block cipher key schedules.

A distinctive aspect of ShieldFS is how it copes with code injection, a common technique used by modern ransomware (as well as other malware). With code injection, a perfectly legitimate process suddenly executes malicious code. Our detection mechanism takes into account both the long- and the short-term history of each process, and of the entire system. Indeed, we are agnostic with respect to how the infection has bootstrapped (e.g., malicious executable, remote code execution) and on the availability of the executable. Rather, we focus on the runtime *effects* on the target system. In fact, as observed in [96], the activity of modern malware can span across multiple process and OS facilities, and, more importantly, an isolated sample to analyze is a luxury in early stage of spreading campaigns. Therefore, detection systems should not assume that a binary executable is available.

We apply our detection approach in a real-time, self-healing virtual filesystem that shadows the write operations. Thus, if a file is surreptitiously altered by one or more malicious processes, the filesystem presents the original, mirrored copy to the user space applications. This shadowing mechanism is dynamically activated and deactivated depending on the outcome of the aforementioned detection logic. Figure 3.1 depicts the logical activity of ShieldFS in comparison with a traditional filesystem.

Experimental Results Summary. We evaluated ShieldFS on 688 samples from 11 distinct families, showing that it can successfully protect user data from real-world attacks performed by recent, state-of-the-art malware families. The system exhibited remarkable accuracy and generalization capabilities even when evaluated via cross-validation on the large dataset that we collected from the 11 real-world machines. Also, we installed ShieldFS on the personal machines in use by 3 volunteers, on which it correctly identified ransomware processes, and successfully reverted their effects. The performance impact of our prototype implementation is such that ShieldFS is applicable in real-world settings.

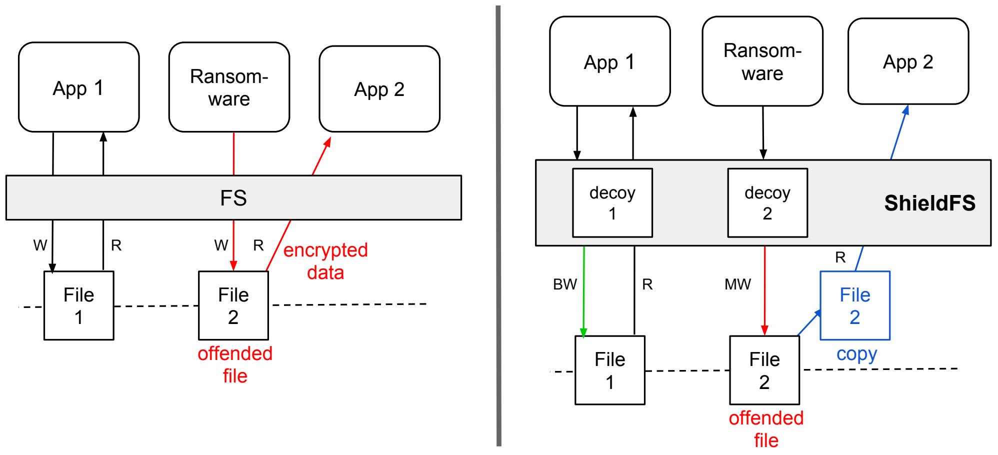


Figure 3.1: On the right ShieldFS shadowing a file offended by ransomware malicious write (MW), in comparison to standard filesystems (on the left).

3.1 Low-Level I/O Data Collection

To understand how ransomware typically interact with the filesystem in comparison to benign applications, the main challenge is to be able to observe them in their usual working conditions (e.g., on a victim's machine). Since there is no such recent data for this purpose, we collected it from real, operational desktop computers for several weeks. First, this provided us with a real-world reference "picture" of how files and folders are organized in a typical computer, which is useful to reproduce an environment that triggers the ransomware activity. Secondly, this approach provided us with a large dataset of filesystem access patterns originating from *benign* applications while exercised by real-user interactions. This is essential to verify whether ransomware and benign applications interact with the filesystem in a significantly different way that could be leveraged for detection.

To carry out our analysis, we developed IRPLogger, a low-level I/O filesystem sniffer, which we installed on real-world machines in use by 11 volunteers. We can categorize the participants as "home," "developer," or "office" users. As summarized in Table 3.1, we collected 28.2 GB of compressed and anonymized data, corresponding to 1,763 million IRPs.

3.1.1 Filesystem Sniffer Details

At the first boot, IRPLogger traverses the directory tree of each mounted drive to collect metadata including total number of files, number of files per extension, and directory depth. The core of IRPLogger is a minifilter driver [47] that intercepts the I/O requests generated for each filesystem primitive invoked by userland code (e.g., `CreateFile`, `WriteFile`, `ReadFile`). IRPLogger enriches the raw IRPs with data including timestamp, writes entropy, and PID. An example log entry (before anonymization) is as follows:

```
<time, program name, PID, IRP op, entropy, file info>
```

When run on the participants' machines, IRPLogger minimizes and hashes any privacy-sensitive data such as the file names and paths. We keep the extension of the accessed

Table 3.1: Statistics of the collected low-level I/O data from 11 real machines during normal usage.

User	Win. ver.	Usage	Data [GB]	#IRPs Mln.	#Procs Mln.	Apps	Period [hrs]	Data Rate [MB/min]
1	10	dev	3.4	230.8	16.60	317	34	7.85
2	8.1	home	2.4	132.1	9.67	132	87	2.04
3	10	office	0.9	54.2	5.56	225	17	0.83
4	7	home	4.7	279.9	18.70	255	122	5.18
5	7	home	2.2	138.1	5.04	141	47	4.10
6	10	dev	1.8	100.4	10.30	225	35	2.42
7	8.1	dev	0.8	49.0	3.28	166	8	5.62
8	8.1	home	0.8	43.9	6.33	148	32	2.16
9	8.1	home	7.7	501.8	24.20	314	215	3.21
10	7	home	0.9	57.6	2.63	151	18	4.60
11	7	office	2.6	175.2	4.69	171	28	8.51
		Total	28.2	1,763.0	107.00	2245	643	-

files in clear, as this detail is needed for computing per-type file statistics and features. Before collection, the logs are split into sessions and compressed for space efficiency.

3.1.2 Ransomware Activity Data Collection

We leveraged IRPLogger also to collect ransomware activity data. During December 2015 we used the VirusTotal Intelligence API to obtain the most recent Windows executables consistently labeled with the main ransomware families (i.e., CryptoWall, TeslaCrypt, Critroni, CryptoDefense, Crowti). We manually ran each sample to ensure that it was fully and properly working (e.g., some samples did not receive instructions and public encryption keys from the attacker's control servers), so obtaining the 383 active ransomware samples summarized in Table 3.2.

Then, we prepared a set of virtual machines on which we activated IRPLogger running on top of Windows 7 (64-bit). We installed common utilities such as Adobe Reader, Microsoft Office, alternative Web browsers, and media players. To create a legitimate-looking system, we included typical user data such as saved credentials, browser history, and realistic decoy files (e.g., images, documents), such that to trigger the samples. We used real files—collected by randomly crawling web search-engines results—reflecting file-type and directory tree distribution of the aforementioned 11 clean machines. At runtime, our analysis environment emulates basic user activity (e.g., moving the mouse, launching applications). Following the best practices for malware experiments suggested by [76], (1) we let the malware executables run for 90

Table 3.2: Statistics of the collected low-level I/O data from 383 ransomware samples.

Ransomware Family	No. Samples	Data	#IRPs Millions
CryptoWall	157 (41.0%)	8.0	286.7
Crowti	125 (32.6%)	5.7	173.1
CryptoDefense	77 (20.1%)	4.5	171.6
Critroni	14 (3.7%)	0.6	3.0
TeslaCrypt	10 (2.6%)	0.9	29.2
Total	383	19.7	663.6

minutes, (2) we allowed the samples to communicate with their control servers, and (3) denied any potentially harmful traffic (e.g., spam) during the experiments. For the sake of scientific repeatability, we are open to provide access to (or the implementation details of) our analysis environment. After each execution, we saved the IRP logs and rolled back each virtual machine to a clean snapshot.

3.1.3 Filesystem Activity Comparison

The remarkable differences in the features distribution shown in Table 3.3 confirms ransomware and benign applications are different filesystem-wise, and motivates us to exploit these results to create a full-fledged remediation system.

We focus our analysis on user data, that is, the main target of ransomware attacks. Contrarily, benign programs, especially system processes (e.g., services, updates manager), access large portions of files in dedicated folders, or in the system folders. For this reason, we separate the IRP logs of user folders from the IRPs of system folders. In practice, we compute the features listed in Table 3.3 twice: first on IRP logs of user paths only (e.g., excluding `WINDOWS` or `Program Files`), and then on *all* paths.

3.2 Approach and Methodology

For clarity, we logically divide our approach into two parts: ransomware activity detection and file recovery. Our file-recovery approach is inspired by copy-on-write filesystems and consists in automatically shadowing a file whenever the original one is modified, as depicted in Figure 3.1. Benign modifications are then asynchronously cleared for space efficiency, and the net effect is that the user never sees the effects of a malicious file encryption.

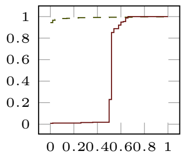
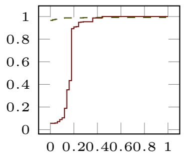
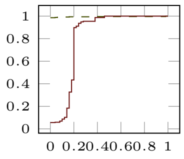
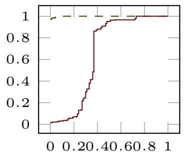
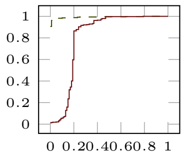
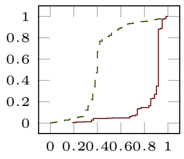
We consider all files as "decoys," that is, we assume that the malware will reveal its behavior because, indeed, it cannot avoid to access the files that it must encrypts to fulfill its goal. The features defined in Table 3.3 summarize the I/O-level activity recorded on these decoys into quantitative indicators of compromise. Thus, the detection and file-recovery parts of our approach are tightly coupled, in the sense that we rely on such decoys to both (1) collect data for detection, and (2) manage the shadowing of the original files.

3.2.1 Ransomware FS Activity Detection

Given the results of our preliminary data analysis in §3.1.3, and the aforementioned assumptions and design decisions, we approach the detection problem as a supervised classification task. Specifically, we propose a custom classifier trained on the filesystem activity features defined in Table 3.3, extracted from a large corpus of IRP logs obtained from clean and infected machines. Once trained, this classifier is leveraged at runtime to decide whether the features extracted from a live system fit the learned feature distributions (i.e., no signs of malicious activity) or not.

Process- and System-centric FS Models. A malware can perform all its malicious actions on a single process, or split it across multiple processes (for higher efficiency and lower accountability). For this reason, our custom classifier adopts several models. One set of models, called *process centric*, each trained on the processes individually. A

Table 3.3: We use these features for both our preliminary assessment (§3.1) and as the building block of the ShieldFS detector (§4.3 §3.3). ShieldFS computes each feature multiple times while monitoring each process, on various portions of filesystem activity, as explained in details in §3.2.1. We normalize the feature values according to statistics of the file system (e.g., total number of files, total number of folders). This normalization is useful to adapt ShieldFS to different scenarios and usage habits. The rightmost column shows a comparison of benign (- - -) vs. ransomware (——) programs by means of the empirical cumulative distribution, calculated on the datasets summarized in Table 3.1 and 3.2, respectively. We notice that ransomware activity is significantly different than that of benign programs according to our features, suggesting that there is sufficient statistical power to tell the two types of programs apart.

Feature	Description	Rationale	Comparison
#Folder-listing	Number of folder-listing operations normalized by the total number of folders in the system.	Ransomware programs greedily traverse the filesystem looking for target files. Although filesystem scanners may exhibit this behavior, we recall that ransomware programs will likely violate multiple of these features in order to work efficiently.	
#Files-Read	Number of files read, normalized by the total number of files.	Ransomware processes must read all files before encrypting them.	
#Files-Written	Number of files written, normalized by the total number of files in the system.	Ransomware programs typically execute more writes than benign programs do under the same working conditions.	
#Files-Renamed	Number of files renamed or moved, normalized by the total number of files in the system.	Ransomware programs often rename files appending a random extension during encryption.	
File type coverage	Total number of files accessed, normalized by the total number of files having the same extensions.	Ransomware targets a specific set of extensions and strives to access *all* files with those extensions. Instead, benign application typically access a fraction of the extensions in a given time interval.	
Write-Entropy	Average entropy of file-write operations.	Encryption generates high entropy data. Although file compressors are also characterized by high-entropy write-operations, we show that the *combined* use of all these features will mitigate such false positives. Moreover, we notice that our dataset of benign applications contains instances of file-compression utilities.	

second model, called *system centric*, trained by considering all the IRP logs as coming from a single, large "process" (i.e., the whole system). The rationale is that the system-centric model has a good recall for multi-process malware, but has potentially more false positives. For this reason, the system-centric model is used only in combination to the process-centric model.

Incremental, Multi-tier Classification. Although our file-recovery mechanism is conservative, we want to minimize the time to decision. Moreover, since the decision can change over time, all processes must be frequently and efficiently monitored. To obtain an acceptable trade off between speed and classification errors we adopt two orthogonal approaches.

First, (1) instead of running our classifiers on the entire available process data, we split the data in intervals, or *ticks*. Ticks are defined by the fraction of files accessed by the monitored process—with respect to the total number of files in the system. In this way, we obtain an array of incremental, "specialized" classifiers, each one trained on increasingly larger data intervals. For instance, when a process has accessed 2% of the files, we query the "2%-classifier" only, and so on. Our experiments (Figure 3.5) show that this technique reduces the #IRPs required to cast a correct detection by three orders of magnitude, with a negligible impact on the accuracy.

Secondly, (2) to account for changes during a process' lifetime, we monitor both the long- and short-term history. In practice, we organize the aforementioned incremental classifiers in a multi-tier, hierarchical structure (as depicted in Figure 3.2), with each tier observing larger spans of data. At each tick, each tier analyzes the data up to N ticks in the past, where N depends on the tier level. We label a process as "ransomware" as soon as at least one of tiers agrees on the same outcome for K consecutive ticks. In §3.4 we show that the choice of K has negligible impact on the false positives.

Example (Code Injection). This example explains how our incremental, multi-tier models handle a typical case. A benign process (e.g., Explorer) is running, and has accessed some files. For the first i ticks ShieldFS will classify it as benign. Now, the Ransomware process injects its code into Explorer's code region. Referring to Figure 3.2, if Ransomware does code injection after the 3rd tick, the global-tier model classifies Explorer as benign, because the long-term feature values are not be affected significantly by the small, recent changes in the filesystem activity of Explorer. Instead, the tier-1 model identifies Explorer as malicious, because the tier-1 features are based *only* on the most recent IRPs (i.e., those occurring right after the code injection). The same applies for tier-2 models after the 4th tick, and so on. If $K = 3$, for instance, and all the triggered tiers agree on a positive detection, the Explorer process is classified as malicious at this point in time. This decision, clearly, can change while more process history is examined.

3.2.2 Cryptographic Primitives Detection

Detecting traces of a cipher within a suspicious process memory, in addition to malicious filesystem activity, is a further indication of its ransomware nature. The malware authors' goal is to efficiently encrypt large sets of files, using a single master key per victim. Thus, instead of relying directly on asymmetric cryptography, which is resource intensive, the strategy is to encrypt each file with a symmetric cipher and a

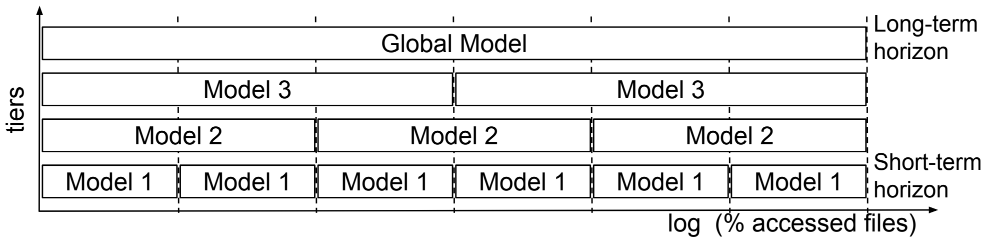


Figure 3.2: Example of the use of incremental models. At each interval, we check simultaneously multiple incremental models at all applicable tiers.

per-file random key, each encrypted with an asymmetric master secret obtained from the attacker's control server.

Efficient Block Ciphers. The most widespread, efficient symmetric-encryption algorithms of choice are fast block ciphers. These ciphers combine the plaintext with a secret key through a sequence of iterations, known as *rounds*. In particular, the key is expanded in a sequence of values, known as the *key schedule*, which is employed to provide enough key material for the combination during all the rounds. Since the key expansion is deterministic and depends on the key alone, it can be pre-computed and reused, with a significant performance gain (e.g., 2 to $4\times$ in case of AES-128). Indeed, all the mainstream cryptographic libraries (e.g., OpenSSL, mBED TLS) and the vast majority of ransomware families do pre-compute the key schedule.

Side Effects. The main side effect of such a pre-computation technique is that the entire key schedule is (and must remain) materialized in memory during all the encryption procedure. We leverage this side effect, and perform a scan of the memory of the running process, checking, at every offset, whether the content of the memory can be obtained as a result of a key schedule computation. Due to the tight constraints present between the key and the expanded key (i.e., sound key schedules impose a bijection between them) it is extremely unlikely that a random sequence of bytes accidentally matches the result of a key expansion, making false positives very unlikely. False negatives may occur if the key schedule is not contiguously stored in memory. However, due to the small size of the involved data (i.e., less than a single 4kiB page), such an event is unlikely to happen due to memory allocation fragmentation.

Note. Although this technique has the benefit of recovering the secret keys used during the encryption, relying exclusively on this criterion for file recovery would not be generic and future-proof: Since each file may be encrypted with a dedicated symmetric key, to guarantee the recoverability of all files, the memory scanning action should be continuous, and there is the risk that some keys are simply missed. Instead, by using our dual approach (i.e., filesystem and memory analysis) ShieldFS can guarantee the recoverability of all files, regardless of how they are encrypted.

3.2.3 Automatic File Recovery Workflow

When ShieldFS is active, any newly created process enters a so-called "unknown" state. Whenever such a process opens a file handle in write or delete mode *for the first time* (only), ShieldFS copies the file content in a trusted, read-only storage area. This storage can be on the main drive or on a secondary drive. In either case, ShieldFS denies access to this area from any userland process by discarding any modification request coming from the upper I/O manager. From this moment on, the process may read or write such file, while ShieldFS monitors its activity. When ShieldFS has collected enough IRPs, the process goes into a "benign," "suspicious," or "malicious" state.

File copies belonging to "benign" processes can be deleted immediately or scheduled for asynchronous deletion, as ShieldFS does. Since storage space is convenient nowadays, leaving copies available for an arbitrarily long time delay does not impose high costs. In turns, it greatly benefits the overall system performance because, by acting as a cache, it limits the number of copy operations required when the same files are accessed (and would need to be copied) multiple times.

For any process that enters the "malicious" state for at least one tick, ShieldFS checks the presence of ciphers within the process. If any are found, it *immediately* suspends the process and restores the offended files. Otherwise, it waits until K positive ticks are reached before suspending the process, regardless of whether a traces of ciphers are found.

Processes can enter a "suspicious" state when the process-centric classifier is not able to cast a decision. In this case, ShieldFS queries the system-centric model. If it gives a positive outcome, then the process enters the "malicious" state. Otherwise the process is classified as "benign."

3.3 ShieldFS System Details

We implemented ShieldFS following the high-level architecture depicted in Figure 3.3, and the detection loop defined in Algorithm 1. We focused on Microsoft Windows because it is the main target of the vast majority of ransomware families. We argue that the technical implementation details may change depending on the target filesystem and OS's internals. However, our approach does not require any special filesystem nor OS support. Thus, we expect that it could be ported to other platforms with modest engineering work.

3.3.1 Ransomware FS Activity Detection

To intercept the IRPs, ShieldFS registers callback functions through the filter manager APIs (i.e., `FltRegisterFilter`). For each IRP, the called function updates the feature values, using separate kernel worker threads for computation-intensive functions (e.g., entropy calculation).

Feature Normalization. To keep the feature values normalized (e.g., number of files read, normalized by the total number of files), the first time the ShieldFS service is run, it scans the filesystem to collect the file extensions, number of files per extensions, and overall number of files.

Since the normalization factors change over time (i.e., new, deleted, or renamed files), ShieldFS updates them in two ways. One mechanism uses a dedicated kernel thread to update the normalization factors *in real time*. This has no performance impact, since ShieldFS already keeps track of the relevant file operations. However, an attacker could exploit it to bias the feature values, by manipulating the normalization factors (e.g., by creating many legitimate, low-entropy files). The second mechanism raises the bar for the attacker because it updates the normalization factors *periodically*

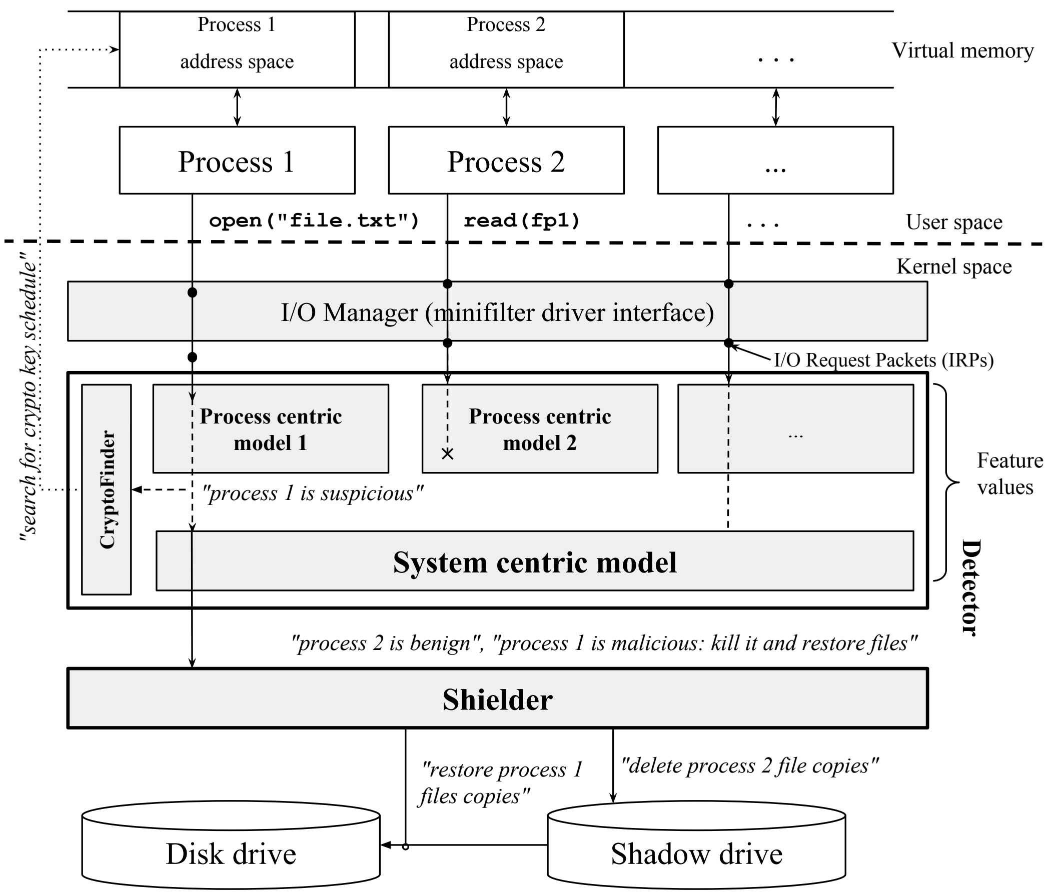


Figure 3.3: High-level overview of ShieldFS. The Detector and the Shielder are Windows minifilter drivers, and the CryptoFinder is kernel driver.

(e.g., once a day). In this way, even if an attacker tries to manipulate our normalization factors, she will need to wait until the next update before starting to access files without triggering any of the features. Although the second mechanism is more resilient to such attacks, it is prone to false positives if users create many files between updates. False positives, however, occur only if a significant number of files are accessed in a way that resembles a ransomware activity (i.e., several folder-listing operations, followed by file reads or renaming, and high-entropy writes). Taking our dataset of benign machines monitored for about a month as a reference, the impact of these false positives is very low compared to the benefits of increased resiliency.

Classifier Details. Each classifier is implemented as a random forest of 100 trees. Each tree outputs either -1 (benign) or $+1$ (malicious). The overall outcome of each process-centric classifier is the sum of its trees' outcome, from -100 (highly benign) to 100 (highly malicious). In case of a tie (i.e., zero), ShieldFS marks the monitored process as "suspicious," and invokes the system-centric classifier to take the decision. In case of a second tie, we conservatively consider the process as malicious.

Monitoring Ticks. ShieldFS gives more relevance to small variations in a feature value when a process has only accessed a few files. At the same time it minimizes the total number of models needed, so as to contain the performance impact. For these reasons, the size of each tick grows exponentially with the percentage of files accessed by a process. After careful evaluation, we used 28 tiers, for intervals ranging from 0.1 to 100%, each one corresponding to a distinct model tier. Adding other ticks beyond 28 would yield no improvements in detection rates, and would instead penalize the performance.

Countermeasure for Buffer-file Abuse. Some versions of Critroni exploit one single file as a write-and-encrypt-buffer. Specifically, the malware moves the target original file in a temporary file, encrypts it, and then overwrites the original file with it. As a result, ShieldFS observes many renaming operations, followed by many read-write operations on a single file, thus biasing the feature values.

To counteract this evasion technique, ShieldFS keeps track of when a file is read (or written) right after being renamed (or moved), such that to update the feature values taking into account the net, end-to-end effect, as if the buffer file was not used. This mechanism comes at no extra cost, since ShieldFS already keeps track of file-renaming operations.

3.3.2 Cryptographic Primitives Detection

ShieldFS checks the memory of processes classified as "suspicious" or "malicious" for the presence of symmetric cryptographic primitives. For the sake of clarity, we remark that the output of CryptoFinder is used as an additional, non-essential feature. Hence, ShieldFS is able to detect even samples that do not show any encryption process, as long as the filesystem activity models are sufficiently (i.e., at least K positive ticks) triggered.

ShieldFS does *not* make any assumption on *how* the cipher is implemented by the malware, save for the materialization of the key schedule. As a proof of concept, we select AES as our target block cipher, due to its widespread use. AES's key schedule expands 128, 192 or 256 key bits into 1408, 1664 or 1920 key schedule bits, respectively. As a consequence, taking all the 2^{64} possible positions in the address space as candidates, and assuming that the accidental occurrence of a key expansion for a location is independent from it occurring for a different one, the probability of a false positive is $2^{64}2^{-1408} = 2^{-1344}$ (in the most favorable case), which is negligible for practical purposes.

CryptoFinder receives the PIDs of suspicious processes by the Detector, through IOCTL. When triggered, CryptoFinder attaches to a process and obtains the list of its memory pages. Specifically, CryptoFinder looks only at the *committed* pages, defined in Windows as the pages for which physical storage has been allocated—either in memory or in the paging file on disk. Then, CryptoFinder runs the key-schedule algorithm on these memory regions and checks whether its expansion occurs. For efficiency reasons, we stop the inspection of a location as soon as there is a single byte mismatch.

3.3.3 Automatic File Recovery

We implemented Shielder as a Windows minifilter driver that monitors file modifications by registering a callback for those `IRP_MJ_CREATE` operations which security context parameter `Parameters.Create.SecurityContext` indicates a "write" or "delete" I/O request. If the target file is not shadowed yet, ShieldFS creates a copy before letting the

Algorithm 1 Detection routine for each process.

```
procedure isRansomware($PID$, $fs\_features$)
    $crypto \leftarrow \bot$
    for $tier \in \{1, ..., top\}$ do
        if $enoughFilesAccessedForTickOf(tier)$ then
            $result \leftarrow \text{ProcessClassifier}_{tier}(fs\_features)$
            $resetFeatureValues(tier)$
            if $result < 0$ then
                $K_{tier} \leftarrow 0$
            else
                $crypto \leftarrow \text{CryptoFinder}(PID)$
                if $result = 0$ then
                    if $\text{SystemClassifier}_{tier}(fs\_features) \geq 0$ then
                        $K_{tier} ++$
                else
                    $K_{tier} ++$
    if $crypto \vee \exists tier : K_{tier} \geq K_{threshold}$ then

        return malicious

    return benign
```

request through. With the same technique it monitors the destination of (potentially malicious) file-renaming operations, by hooking the `IRP_MJ_SET_INFORMATION` requests having the `ReplaceIfExists` flag set. File handing and indexing in the shadow drive is based on the `FILE_ID` identifier assigned by NTFS to each file.

Transaction Log. ShieldFS maintains a transaction log of the relevant IRPs (e.g., those resulting from file modifications). Whenever a process is classified as malicious, ShieldFS inspects such log and restores each file affected by the offending process.

File copies are deleted only when the processes that modified the original file have been cleared as "benign." ShieldFS treats the shadow drive as a cache: It avoids shadowing the same file if a fresh copy (i.e., not older than T hours) already exists. According to our experiments, based on the workload of real-world users (obtained form our large-scale data collection), the age T imposes acceptable overhead (below 1%) and can be safely set to any number between 1 and 4. In §3.5 we discuss how the choice of T raises the bar for the attacker who wants to successfully encrypt a large portion of files.

Whitelisting of Support Files. Files that have no value for a user are of no interest for ransomware attacks. An example are application-support directories, which contain cache or temporary files, which are frequently accessed by benign applications. These folders can be safely whitelisted to reduce the performance overhead due to the frequent operations on such files. To avoid that an attacker could exploit the whitelisted folders as a "demilitarized zone" where to copy the target files (prior to encrypting them), we adopt the following solution. Any process that has *never* accessed a whitelisted folder is considered "suspicious" as soon as it attempts to move files into it. The files offended by this operation are preemptively shadowed.

Windows Shadow Copy. Recent Windows versions have a so-called Volume Shadow Copy Service. However, Windows shadow copies have two issues. First, the copies are created only during the next power down and boot cycle. Instead, as we already mentioned, our approach is designed for short-term backup that can allows users to restore recently modified files. Secondly, shadow copies can be easily bypassed and deleted, as most of recent ransomware families do before starting the encryption process [53].

3.4 Experimental Results

As we did for our preliminary analysis (§3.1.2), we evaluated ShieldFS on an analysis environment with virtual machines provisioned so as to mimic the file content and organization of potential victim machines.

We first performed a thorough *cross validation* to assess (1) the generalization capabilities of our classifiers, and (2) the impact of the parameter choice on the overall detection quality and performance. Second, we infected physical machines *in use by real users* (for their day-to-day activities) with 3 samples of ransomware families. ShieldFS was able to detect their activity and fully recover all the compromised files. Third, we evaluated the *detection and file-recovery capabilities* against ransomware samples that ShieldFS has never seen before. Last, we measured the performance overhead of ShieldFS by considering the typical usage workload, where "typical" refers to our initial large-scale collection of I/O filesystem logs.

A video demo of ShieldFS in action is available on YouTube at [5].

3.4.1 Detection Accuracy

Cross validation allows to reveal the presence of overfitting-induced biases and thus is a crucial aspect of any machine-learning-based approach. We conducted three cross-validation experiments to evaluate the quality of the Detector on our dataset of 383 ransomware samples and 2,245 benign applications from the 11 user machines. We count positive or negative detections at the process granularity, and calculate the TPR and FPR based on the true overall number of benign and ransomware processes.

10-fold Cross Validation. We calculated the true- and false-positive rate on 10 random train/test splits. Figure 3.4 and 3.5 show the TPR and FPR in function of the minimum percentage of files, and #IRPs, respectively, needed to cast the decision. The results show the benefit of the system-centric model as a tie breaker, and the incremental approach as an early detector, which requires orders of magnitude less IRPs to cast a decision, with almost no impact on the FPR (i.e., from 0.0 to 0.00015 in the worst case).

One-machine-off Cross Validation. To further show the independence of our detection results from the specific machine that generates the benign subset of training and testing data, we performed a per-machine cross validation. We selectively removed the data of one machine from the training set, and used it as the testing set. We repeated this procedure for each of the 11 machines.

Table 3.4 shows (1) that ShieldFS has no strong dependency from the training-testing data split, and (2) confirms that the system-centric model is useful to reduce FPs by acting as a tie breaker.

Causes of False Positives. We found only two cases of false positives. For the first user machine, the detector triggered because `explorer.exe` biased the normalization,

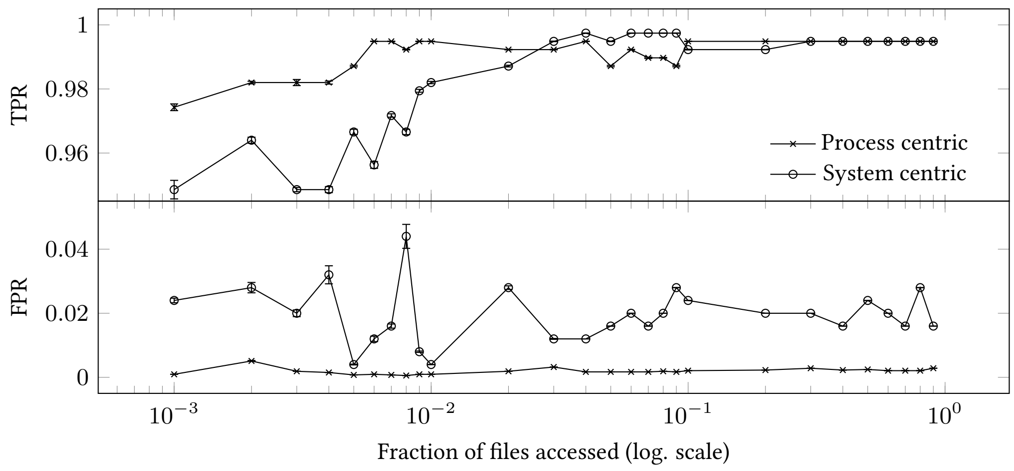


Figure 3.4: 10-fold Cross Validation: Average and standard deviation of TPR and FPR with process- vs. system-centric detectors.

by accessing a very large number of files (more than the normalization factors, which were not up to date). This motivated us to implement the mechanism that live-updates the system-wide, feature counts for normalization (rather than doing such an update periodically). This *eliminates the false positives*, creating however a small opportunity for the attacker to bias the normalization factors. This trade off is clearly inherent in the statistical nature of ShieldFS.

Interestingly, in 4 out of 11 machines we found activity of the WinRar file-compression utility, which performed high-entropy writes. Nevertheless, WinRar was correctly classified as benign, thanks to the contribution of the remainder features.

The second false positive was Visual Studio, which wrote 175 files, with a very high average entropy (0.948). This was an isolated case, which happened only on one of the 32 Visual Studio session recorded in our dataset.

Parameter Setting. The choice of K, the number of consecutive positive detections required to consider a process as malicious, can be set to minimize the FPR to zero, at the price of a very small variation (within +/-0.5%) of TPR. Or vice versa. Table 3.5 shows that setting $K = 3$ maximizes the TPR, with very few false positives. Instead, with $K = 6$, ShieldFS did not identified a sample that performed injection into a be-

Table 3.4: FPR with One-machine-off Cross Validation

User	**FPR** [%]		
Machine	Process	System	**Outcome**
1	0.53	23.26	**0.27**
2	0.00	0.00	**0.00**
3	0.00	0.00	**0.00**
4	0.00	1.20	**0.00**
5	0.22	45.45	**0.15**
6	0.00	4.76	**0.00**
7	0.00	88.89	**0.00**
8	0.00	0.00	**0.00**
9	0.00	0.00	**0.00**
10	0.00	0.00	**0.00**
11	0.00	0.00	**0.00**

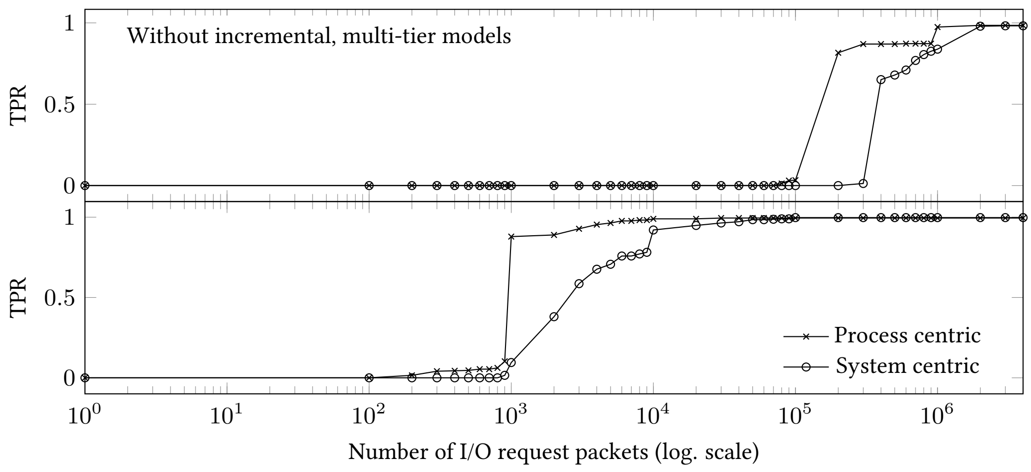


Figure 3.5: 10-fold Cross Validation: TPR of process- and system-centric detectors, with and without the incremental, multi-tier approach. FPR ranges from 0.0 to 0.0015.

nign process and that encrypted files very slowly. Generally, false negatives are more expensive than false positives in ransomware-detection problems, thus we advise for values of K that maximize the TPR. This has the additional benefit of reducing the number of IRPs required for a correct detection.

3.4.2 Protection of Production Machines

In order to evaluate our system in real scenarios, we tested ShieldFS on three distinct real machines (running Windows 7 and 10), in use by real users for their day-to-day activities for years, containing 2,319, 165,683, and 144,868 files, respectively. We randomly selected 3 samples[1] from our dataset (Critroni, TeslaCrypt, and ZeroLocker) and manually analyzed them to ensure that they were not stealing any personal information. After cloning the hard drives as a precaution, we installed ShieldFS, and infected the machines. All the three samples were correctly detected and all the affected files were correctly restored automatically.

3.4.3 Detection and Recovery Capabilities

We setup an environment as described in §3.1.2, with dummy files to reproduce a real-user setting. Moreover, we stored 9,731 files typically targeted by ransomware attacks (e.g., images and documents of various formats), of which we pre-calculated

[1] e89f09fdded777ceba6412d55ce9d3bc, 209a288c68207d57e0ce6e60ebf60729, bd0a3c308a6d3372817a474b7c653097

Table 3.5: 10-fold Cross-Validation: Choice of K.

K	FPR	TPR	IRPs
1	0.208%	100%	35664
2	0.076%	100%	43822
3	**0.038%**	**100%**	**67394**
4	0.019%	99.74%	80782
5	0.019%	99.74%	104340
6	0.000%	99.74%	135324

the MD5 for integrity verification after each experiment. We then trained ShieldFS on the large dataset of IRP logs collected as part of our preliminary analysis.

Dataset of Unseen Samples. In addition to the cross-validation experiments on 383 samples, which already show the predictive and genralization capabilities of ShieldFS, we obtained 305 novel, working ransomware samples and let them run for 60 minutes on the machines protected by ShieldFS. This dataset (Table 3.6) is completely disjoint from the training dataset and was collected from VirusTotal as of May 2016. Interestingly, seven families (Locky, CryptoLocker, TorrentLocker, DirtyDecrypt, PayCrypt, Troldesh, ZeroLocker) are not present in the training dataset.

Detection of Unseen Samples. ShieldFS prevented malicious encryption in 100% of the cases, by restoring the 97,256 compromised files, and correctly detected 298 (97.70%) of the samples without any false positive. The top-tier, process-centric model contributed to detecting 95.2% of the samples, the incremental models were effective mainly in the case of ransomware performing code injections (4.3%), as expected. In one case, the incremental process-centric models identified the malicious process as suspicious and ShieldFS invoked the system-centric model to take a final decision. CryptoFinder contributed to the detection of 69.3% of the samples.

Causes of False Negatives. Seven samples remained inactive for most of our analysis and encrypted just few files (less than 30). Fortunately, thanks to our conservative file-shadowing strategy, ShieldFS had copied the original files, allowing their recovery. We investigated the cause of false negatives in the detection of cryptographic primitives and we found no evidence showing that the remaining samples were using AES. Therefore, we conclude that CryptoFinder's detection capability of AES key schedule is 100%.

3.4.4 System Overhead

We evaluated the performance overhead and additional storage space requirements of ShieldFS.

User-Perceived Overhead. Our goal is to quantify, with good approximation, how much would ShieldFS slow down the typical user's tasks, on average. To this end, we

Table 3.6: Dataset of 305 unseen samples of 11 different ransomware families.

Ransomware Family	No. Samples	Detection Rate
Locky	154 (50.5%)	150/154
TeslaCrypt	73 (23.9%)	72/73
CryptoLocker	20 (6.6%)	20/20
Critroni	17 (5.6%)	17/17
TorrentLocker	12 (3.9%)	12/12
CryptoWall	8 (2.6%)	8/8
Troldesh	8 (2.6%)	7/8
CryptoDefense	6 (2.0%)	5/6
PayCrypt	3 (1.0%)	3/3
DirtyDecrypt	3 (1.0%)	3/3
ZeroLocker	1 (0.3%)	1/1
Total	305	298/305

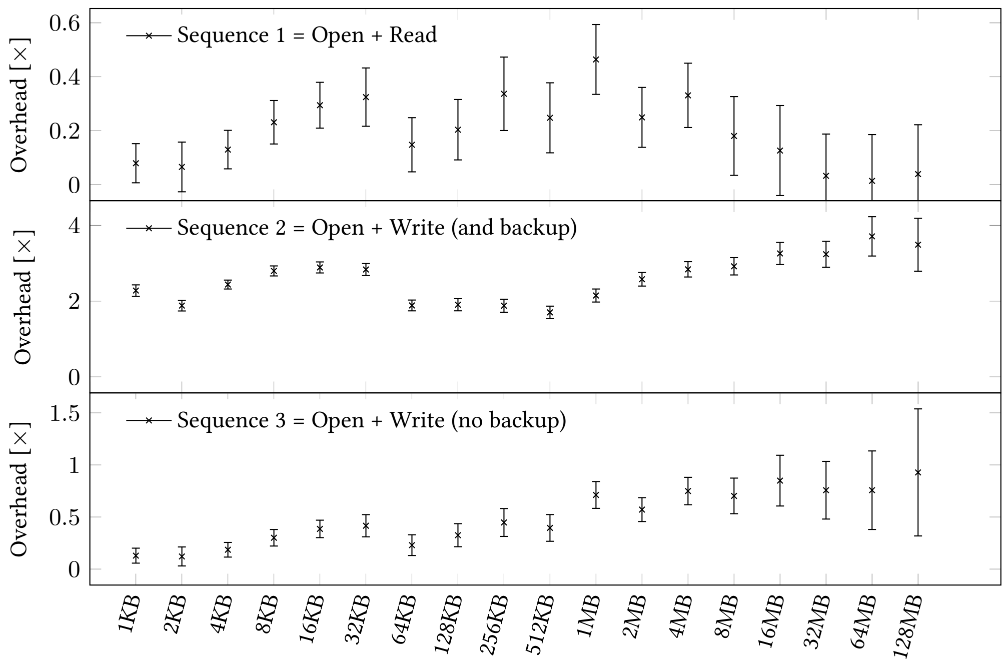


Figure 3.6: Micro Benchmark: Average overhead.

distributed to 5 users a new version of IRPLogger that collected file-size information in addition to the usual IRP logs. Then, we reconstructed 6 hours worth of sequences of high-level system calls by analyzing about one month of low-level IRPs. For example, one `IRP_MJ_CREATE` followed by one or more `IRP_MJ_READ` corresponds to a `FileRead` call, and so on, by abstraction. Then, we estimated the perceived overhead for a user-level task as the average overhead due to all the filesystem calls executed during such task, taking into account the size of the affected files. We fixed 10 minutes as the duration of a user-level task, that is, while the user is interacting with the computer uninterruptedly. Figure 3.7 shows that the average estimated overhead is $0.26\times$. Indeed, we barely perceived it while using a machine protected by ShieldFS.

Runtime Overhead: Micro Benchmarks. We also evaluated the in-the-small performance impact of ShieldFS. We considered three sequences of filesystem operations on a series of 1,800 files of 18 varying sizes (from 1 KB to 128 MB): (1) open and read the files, (2) open and write them when they are not backed up already, and (3) open and write them when they are already backed up. We run each sequence 100 times on a Windows 10 machine equipped with a rotational hard disk drive, with and without ShieldFS, rebooting the machine after each test to avoid caching side effects.

Table 3.7: Measured storage space requirements on real-users machines ($T = 3h$) and cost estimation considering $3¢/GB.

User	Period	Storage	Required	Storage	Overhead	Max Cost
	[hrs]	Max [GB]	Avg. [GB]	Max [%]	Avg [%]	[USD]
1	34	14.73	0.63	4.29	0.18	44.2¢
2	87	0.62	0.19	0.95	0.29	1.86¢
4	122	9.11	0.73	8.53	0.68	27.3¢
5	47	2.41	0.56	5.49	1.29	7.23¢
7	8	1.00	0.39	3.35	1.28	3.00¢

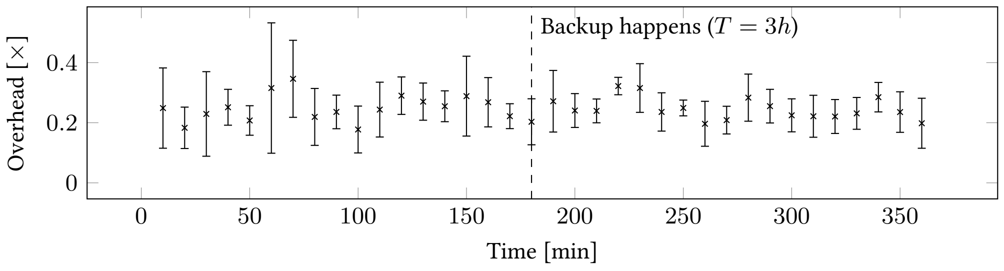


Figure 3.7: Average (and standard deviation) perceived overhead introduced by ShieldFS on 5 real-users machines.

Figure 3.6 shows the overhead of each sequence. The overhead is higher (1.8–$3.8\times$) when files need to be backed up, and remarkably lower (0.3–$0.9\times$) when files are already backed up.

Storage Space Requirements. During our experiments we kept track of the storage space required by ShieldFS to keep secure copies. Table 3.7 shows that with $T = 3h$, in the worst case (i.e., all files need to be backed up within T), ShieldFS requires 14.73 GB of additional storage space (i.e., $44.2¢).

Parameter Setting. The T parameter determines how often ShieldFS creates copies of the files that require to be shadowed. Table 3.8 shows the average overhead and storage space required for $T \in [1, 4]$ hour(s) measured during our experiments. We can conclude that T does not significantly influence the overall performance overhead. Thus, as further discussed in §3.5, we advise to set it as high as to match the on-premise, long-term backup schedule.

3.5 Discussion of Limitations

From the results of our experiments we discuss the following list of limitations, in decreasing order of importance.

Susceptibility to Targeted Evasion. Ticks are essentially the "clock" of ShieldFS. At each tick, a decision is made. Since ticks are not based on time, but on the percentage of files accessed, an adversary may be interested in preventing to trigger the ticks, so to avoid detection. However, the only way to do it is to access zero or very few files, which is clearly against the attacker's goal. Alternatively, in order not to cause a significant change in the feature values after code injection, an adversary may try to find an existing, *benign* host process that has *already* accessed about as many files as the attacker wants to encrypt. This is very unlikely because, by design, such process can exist only if it has not already triggered the detection (otherwise ShieldFS would have already killed it already). That is, only if it has accessed a large number of files

Table 3.8: Influence of T on runtime and storage overhead.

T	Runtime	Overhead	Storage	Overhead		
[hrs]	Avg [×]	Std.dev [×]	Max [GB]	Avg [GB]	Max [%]	Avg [%]
1	0.263	0.0404	5.4838	0.4040	4.353	0.586
2	0.262	0.0404	5.8402	0.4875	4.762	0.720
3	0.261	0.0403	5.5768	0.4994	4.522	0.746
4	0.260	0.0403	5.5927	0.5150	4.545	0.766

without violating the other features (e.g., mainly read operations, low entropy files). Assuming that the malware can find such a benign process to inject its malicious code, the process' features will start to change as soon as the malicious code will start encrypting the aforementioned files. At some point, the malicious code cannot avoid performing many write operations of high-entropy content.

If the malware knows precisely the thresholds of the classifiers and value of the parameter T, it could attempt to perform a mimicry attack [93] encrypting few files so as to remain below the thresholds until T hours. In this way, it will be identified as benign and the victim will loose the original copies. However to remain unaccountable, a ransomware cannot encrypt all the files in one round, so it would need to repeat this procedure every T hours. Setting T to large values will raise the bar, by forcing the attacker to wait for long. On the other hand, setting T very low guarantees that the recent (benign) modifications are accounted in the secondary drive. In this way, if a restore is needed, a very recent (up to T) copy is available. In other words, T allows to trade off mimicry resilience versus data freshness.

Multiprocess Malware. Ransomware injecting malicious code into many benign processes, each of them performing a small part of the malicious activity, could evade our detector if the attacker knows the feature values—which, is challenging for a userland malware. Multiprocess malware is partially mitigated by the combination of system-centric models with the incremental, multi-tier strategy. Nevertheless, ransomware could perform encryption very slowly. This however, is against the attackers' goal, who wants to encrypt all files before users can notice any change. Last, even if a malicious process is *not* detected, thanks to our conservative file-shadowing approach, a user noticing the encrypted files can manually restore the original files from the copies.

Cryptography Primitives Detection Evasion. A possible cause of false negatives of our approach is the use of dedicated ISA extensions of modern CPUs (e.g., Intel AES-NI [44]) to perform the encryption off memory, using a dedicated register file. However, in such case the malware binary code would contain those specific instructions, not to mention that the malware will work only if the victim machine supports the Intel AES-NI extensions.

The current proof-of-concept implementation of ShieldFS supports only the detection of AES. Supporting other ciphers is an implementation effort, as our approach is valid for the majority of symmetric block ciphers.

Tampering with the Kernel. ShieldFS runs in a privileged kernel mode. We implemented ShieldFS to be "non unloadable" at runtime, even by administrator users. Furthermore, ShieldFS is able to deny any operation that attempts to delete or modify the driver binaries. An administrator-privileged process, however, could try to prevent ShieldFS service from starting at boot, by modifying the Windows registry, and force a reboot. This limitation can be mitigated by embedding our approach directly in the kernel without the need for a service. Doing so, the only chance to bypass our system is to compromise the OS kernel.

Preventing Denial of Service. A malware could attempt to compromise ShieldFS itself by filling up the shadow drive. First, in this scenario it is likely that ShieldFS detects and stops the malicious process before it fills the entire space. Second, ShieldFS makes the shadow drive read-only, denying any modification request coming from userland processes. Last, ShieldFS could monitor the shadow drive and alert the user when it is running out of space.

3.6 Related Works

Kharraz et al. [51] studied the behavior of scareware and ransomware, observing its evolution during the last years, in terms of encryption mechanisms, filesystem interactions, and financial incentives. They suggested some potential defenses, but evaluating them was out of the scope of their paper. Indeed, while [51] analyzed the filesystem activity of ransomware, the authors (and any other work) did not focus on analyzing the filesystem activity of *benign* applications, which we found crucial to build a robust detector.

Concurrently and independently to our work, Kharraz et al. [49] and Scaife et al. [79] published two ransomware detection approaches, respectively UNVEIL and CryptoDrop. Although they both look at the filesystem layer to spot the typical ransomware activity, they do not provide any recovery capability. Also, their approaches do not include identification of cryptographic primitives. Differently from our work, UNVEIL includes text analysis techniques to detect ransomware threatening notes and screen lockers, along the line of [9], and CryptoDrop uses similarity-preserving hash functions to measure the dissimilarity between the original and the encrypted content of each file. These two techniques are complementary to ours, and can be added to ShieldFS as additional detection features.

Andronio et al. [9] studied the ransomware phenomenon on Android devices, proposing an approach, HelDroid, to identify malicious apps. Besides the difference in the target platform, HelDroid looks at how ransomware behaves at the application layer, whereas we focus on its low-level behavior. Thus, their approach is complementary to ours, also because it is based on static analysis.

Our data-collection and mining phase is somehow akin to what Lanzi et al. [56] did to perform a large-scale collection of system calls, with the purpose of studying malware behavior by means of the system and API call profiles. We focus on IRPs instead as they better capture ransomware behavior.

Lestringant et al. [58] applied graph isomorphism techniques to data-flow graphs in order to identify cryptographic primitives in binary code. Although [58] works at binary level, whereas ShieldFS identifies usage of cryptographic primitives at runtime, it is a valid alternative that can be used to complement our CryptoFinder.

3.7 Concluding Remarks

In this chapter, we presented an approach to make modern operating systems more resilient to malicious encryption attacks, by detecting ransomware-like behaviors and reverting their effects safeguarding the integrity of users' data.

We foresee ShieldFS as a countermeasure that keeps an always-fresh, automatic backup of the files modified in the short term. We argue that, although older files can be asynchronously backed up with on-premise systems (because they have less strict time constraints), recent files may be of immense value for a user (e.g., time-sensitive content); even the loss of a small update to an important file may end up in the decision to pay the ransom, because the existing backup is simply too old. With traditional backup solutions alone there exist a trade off between performance, space and "freshness," not to mention that a ransomware may encrypt the backups as well! Generally, traditional solutions work well for incremental backups, long-term archives with no real-time constraints. Pushing such backup solutions to tighter time constraints while keeping reasonable system performance may result in side effects. For instance, once a file is encrypted by a ransomware, there exists a risk that it may replace the plaintext backup. Instead, ShieldFS works at a lower level. Thus, it is transparent to a ransomware that works at the filesystem's logical view. Therefore, it is best suited for the protection of short-term file changes, leaving traditional backups protecting from long-term file changes.

4. Detecting Obfuscated Privacy Leaks in Mobile Applications

One main concern of mobile app users is the leakage of private information: Mobile apps, and third-party advertisement (ad) libraries in particular, extensively collect private information and track users in order to monetize apps and provide targeted advertisements. In response, the security community has proposed numerous approaches that detect whether a given app leaks private information to the network or not. The majority of approaches utilize data flow analysis of the app's code, both through static and/or dynamic taint analysis. Tools based on static taint analysis, such as FlowDroid [11], identify possible sources of private information and determine how their values flow throughout the app and, eventually, to sinks, such as the network. Dynamic taint analysis tools, such as TaintDroid [29], execute apps in an instrumented environment and track how private information is propagated while the app is running. Finally, AppAudit combines both approaches, determining critical flows that leak data through static analysis and verifying them through an approximated dynamic analysis [101].

While these tools provide useful insights, they suffer from several limitations that affect their adoption, especially when the threat model considers apps that try to hide the fact that they are leaking information. Adversaries can deliberately add code constructs that break the flow of information throughout an app and make data flow analysis approaches "lose track" of tainted values. Related works [19, 77] have already demonstrated how an app can, for example, use indirections through implicit control flows or through the file system to efficiently bypass static and dynamic data flow analysis. Furthermore, static and dynamic analysis approaches for mobile apps usually only inspect data flow in Dalvik bytecode (i.e., the Java component of the app) and miss data leaks in native code components, which are becoming more and more prevalent [8, 61]. Both static and dynamic analysis can also have false positives, mainly due to a phenomenon called overtainting: imprecisions in modeling how information flows through specific instructions, or imprecisions introduced to make the analysis scalable might establish that a given value is "tainted" with private information even when, in fact, it is not.

Since static analysis does not perform real-time detection of privacy leaks, and dynamic analysis requires heavy instrumentation, and is thus more likely to be used by app stores than by end users, researchers have recently proposed a more light-weight alternative: identifying privacy leaks on the network layer through traffic interception [62, 82, 57, 73, 74]. However, obfuscation is out-of-scope for the majority of approaches as they perform simple string matching and essentially "grep" for hardcoded values of private information and some well-known encodings such as Base64 or standard hashing algorithms. ReCon [74] is the most resilient to obfuscation as it identifies leaks based on the structure of the HTTP requests, for example by learning that the value following a "deviceid" key in a HTTP GET request is probably a device ID. Still, the underlying machine learning classifier is limited by the data it is trained on, which is collected through TaintDroid and manual analysis—if the labelling process misses any leak and its corresponding key, e.g., due to obfuscation or custom encoding, ReCon will not be able to detect it.

In general, the transformation of privacy leaks, from simple formatting and encoding to more complex obfuscations, has gotten little attention so far. Only BayesDroid [91] and MorphDroid [37] have observed that the leaked information does not always exactly match the original private information, but focused on leaks consisting of subsets or substrings of information instead of obfuscation. It is unclear to what extent apps can hide their information leaks from state-of-the-art tools. For this purpose, we developed a novel automatic analysis approach for privacy leak detection in HTTP(S) traffic that is agnostic to how private information is formatted or encoded. Our work builds on the idea of observing network traffic and attempts to identify leaks through a technique similar to the differential analysis approach used in cryptography: first, we collect an app's network traffic associated with multiple executions; then, we modify the input, i.e., the private information, and look for changes in the output, i.e., the network traffic. This allows us to detect leaks of private information even if it has been heavily obfuscated.

The idea to perform differential black-box analysis is intuitive, and in fact, has already been explored by Privacy Oracle [48] for the detection of information leaks in Windows applications. One of the main challenges of performing differential analysis is the elimination of all sources of non-determinism between different executions of an app. Only by doing this one can reliably attribute changes in the output to changes in the input, and confirm the presence of information leaks. While Privacy Oracle was mainly concerned with *deterministic executions* to eliminate OS artifacts that vary between executions and could interfere with the analysis, we observed that *non-deterministic network traffic* poses a far greater challenge when analyzing modern apps. Due to the frequent use of random identifiers, timestamps, server-assigned session identifiers, or encryption, the network output inherently differs in every execution. These spurious differences make it impractical to detect any significant differences caused by actual privacy leaks by simply observing variations in the raw network output.

One key contribution of this work is to show that, in fact, it is possible to explain the non-determinism of the network behavior in most cases. To this end, we conducted a small-scale empirical study to determine the common causes of non-determinism in apps' network behavior. Then, we leveraged this knowledge in the development of a new analysis system, called AGRIGENTO, which eliminates the root causes of non-determinism and makes differential analysis of Android apps practical and accurate.

Our approach has the key advantage that it is "fail-safe": we adopt a conservative approach and flag any non-determinism that AGRIGENTO cannot eliminate as a "potential leak." For each identified leak, AGRIGENTO performs a risk analysis to quantify the amount of information it contains, i.e., its risk, effectively limiting the channel capacity of what an attacker can leak without raising an alarm. We performed a series of experiments on 1,004 Android apps, including the most popular ones from the Google Play Store. Our results show that our approach works well in practice with most popular benign apps and outperforms existing state-of-the-art tools. As a result, Agrigento sheds light on how current Android apps obfuscate private information before it is leaked, with transformations going far beyond simple formatting and encoding. In our evaluation, we identified several apps that use custom obfuscation and encryption that state-of-the-art tools cannot detect. For instance, we found that the popular InMobi ad library leaks the Android ID using several layers of encoding and encryption, including XORing it with a randomly generated key.

It is not surprising that developers are adopting such stealth techniques to hide their privacy leaks, given the fact that regulators such as the Federal Trade Commission (FTC) have recently started to issue sizable fines to developers for the invasion of privacy of their users [34, 35]: aforementioned InMobi for example is subject to a penalty of $4 million and has to undergo bi-yearly privacy audits for the next 20 years for tracking users' location without their knowledge and consent [36]. Also, counterintuitively to the fact that they are collecting private information, app developers are also seemingly becoming more privacy-aware and encode data before leaking it. Unfortunately, it has been shown that the structured nature of some device identifiers makes simple techniques (e.g., hashing) not enough to protect users' privacy [28, 38]. Consequently, on one hand there is a clear motivation for developers to perform obfuscation—either to maliciously hide data leaks, or to secure user data by not transmitting private information in plaintext—on the other hand privacy leak detection tools need to be agnostic to any kind of obfuscation.

4.1 Motivation

This section discusses a real-world example that motivates our work. Consider the snippet of code in Figure 4.1. The code first obtains the Android ID using the Java Reflection API, hashes the Android ID with SHA1, XORs the hash with a randomly generated key, stores the result in JSON format, and encrypts the JSON using RSA. Finally, it sends the encrypted JSON and the XOR key through an HTTP POST request.

Depending on how this functionality is implemented, existing tools would miss the detection of this leak. Existing approaches based on static analysis would miss this privacy leak if the functionality is implemented in native code [8], dynamically loaded code [69], or in JavaScript in a WebView [63]. Furthermore, the use of the Java Reflection API to resolve calls at runtime can severely impede static analysis tools.

More fundamentally, the complex lifecycle and component-based nature of Android apps make tracking private information throughout an app extremely challenging, and both static and dynamic approaches are sensitive to the disruption of the data flow. For instance, many existing tools would miss this leak if this functionality is implemented in different components. Similarly, if the app first writes the private information to a file, e.g., its settings, and only later reads it from there to transmit it via a network sink, any data flow dependency would be lost. Furthermore, data flow is also lost when the implementation is incomplete and fails to propagate data flows through relevant functions: TaintDroid for example does not track data flows through hashing functions [71].

```
StringBuilder json = new StringBuilder();
// get Android ID using the Java Reflection API
Class class = Class.forName("PlatformId")
String aid = class.getDeclaredMethod("getAndroidId",
       Context.class).invoke(context);
// hash Android ID
MessageDigest sha1 = getInstance("SHA-1");
sha1.update(aid.getBytes());
byte[] digest = sha1.digest();
// generate random key
Random r = new Random();
int key = r.nextint();
// XOR Android ID with the randomly generated key
byte[] xored = customXOR(digest, key);
// encode with Base64
String encoded = Base64.encode(xored);
// append to JSON string
json.append("O1:\'");
json.append(encoded);
json.append("\'");
// encrypt JSON using RSA
Cipher rsa = getInstance("RSA/ECB/nopadding");
rsa.init(ENCRYPT_MODE, (Key) publicKey);
encr = new String(rsa.doFinal(json.getBytes()));
// send the encrypted value and key to ad server
HttpURLConnection conn = url.openConnection();
OutputStream os = conn.getOutputStream();
os.write(Base64.encode(encr).getBytes());
os.write(("key=" + key).getBytes());
```

Figure 4.1: Snippet of code leaking the Android ID using obfuscation and encryption. The example is based on real code implemented in the popular InMobi ad library.

Existing black-box approaches that analyze the network traffic would miss the detection of this leak as well, as they only consider basic encodings, such as Base64 or standard hashing algorithms, and cannot handle complex obfuscations techniques that combine multiple different encodings and algorithms such as the example code in Figure 4.1.

Our work attempts to fill this gap: we focus on designing and developing an approach able to detect privacy leaks even when custom obfuscation mechanisms are used. Our approach is black-box based, so it is not affected by code obfuscation or complex program constructs. Furthermore, our approach can handle obfuscations of the actual data since it does not look for specific tokens that are known to be associated with leaks, but rather treats *every* inexplicable change in the network traffic as a potential leak.

We stress that the example we discussed in this section is not synthetic, but it is actually the simplified version of a snippet taken from one of the most popular apps in the Google Play Store. Specifically, this example is the simplified version of a functionality implemented in the popular InMobi ad library. We also note that this case of nested encodings and encryption is not just an isolated example: our experiments, discussed at length in §4.5, show that these obfuscated leaks occur quite frequently and that existing black-box approaches are unable to detect them.

4.2 Sources of Non-Determinism

One of the key prerequisites for performing differential analysis is to eliminate any sources of non-determinism between different executions. Only by doing so, one can reliably attribute any changes in the network output following changes in private input values to information leakage. While previous work has focused on deterministic executions through the use of OS snapshots [48], according to our experiments the network itself is by far the largest source of non-determinism.

When executing an app multiple times on exactly the same device, with the same settings, and using the same user input, one would intuitively expect an app to produce exactly the same (i.e., deterministic) network traffic during every execution. However, our preliminary experiments showed that this is not the case: the network traffic and more specifically the transmitted and received data frequently changes on every execution, and even between the same requests and responses during a single execution.

This non-determinism is not necessarily something that is introduced by the app developer intentionally to evade analysis systems, but, instead, it is most often part of the legitimate functionality and standard network communication. We conducted a small-scale study on 15 Android apps randomly selected from the Google Play Store, and we investigated the most common sources of non-determinism in network traffic. We were able to identify the following categories:

- **Random values**. Random numbers used to generate session identifiers or, for instance, to implement game logic. Also, the Android framework provides developers with an API to generate 128-bit random universally unique identifiers (UUID). In the most common scenario, apps use this API to generate an UUID during the installation process.
- **Timing values**. Timestamps and durations, mainly used for dates, logging, signatures, and to perform measurements (e.g., loading time).
- **System values**. Information about the state and the performance of system (e.g., information about free memory and available storage space).
- **Encrypted values**. Cryptographic algorithms use randomness to generate initialization vectors (IV) and padding.
- **Network values**. Information that is assigned by a network resource (e.g., cookies, server-assigned session identifiers).
- **Non-deterministic execution**. Randomness inherent to the execution of an app, such as different loading times affecting the UI exploration.

4.3 Approach

For any given app, our analysis consists of two main phases. In the first phase (see §4.3.1), called *network behavior summary extraction*, we execute the app multiple times in an instrumented environment to collect raw *network traces*, and *contextual information*, which allows us to attribute the non-determinism that we see in the network behavior to the sources discussed in §4.2. We then combine these network traces with the contextual information to create a *contextualized trace* for each run. Finally, we merge the contextualized network traces of all runs into a *network behavior summary* of the app.

In the second phase of our approach (see §4.3.2), we run the app again in exactly the same instrumented environment, with the only difference that we change one of the input sources of private information (e.g., IMEI, location). We then compare the contextualized trace collected in this final run with the network behavior summary of the previous runs to identify any discrepancy. We perform this comparison in two steps: *differential analysis*, which identifies differences, and *risk analysis*, which scores the identified differences to determine potential privacy leaks.

Figure 4.2 shows a high-level overview of our approach, while Figure 4.3 illustrates the individual steps in more detail using a simplified example.

4.3.1 Network Behavior Summary Extraction

Network Trace & Contextual Information. For each execution of the app in our instrumented environment, we collect a network trace, which contains the raw HTTP flows generated by the app, and *contextual information*, which contains the values generated by any of the sources of non-determinism we described earlier. Our approach here goes beyond simple network traffic analysis, and includes instrumenting the way the app is interacting with the Android framework. Specifically, Agrigento is able to eliminate the different sources of non-determinism by intercepting calls from the app to certain Android API calls and recording their return values, and in some cases replacing them—either by replaying previously seen values or by returning constant values. First, Agrigento records the timestamps generated during the first run of each app, and replays the same values in the further runs. Second, it records the random identifiers (UUID) generated by the app. Third, it records the plaintext and ciphertext values whenever the app performs encryption. Finally, the instrumented environment sets a fixed seed for all random number generation functions, and replaces the values of system-related performance measures (e.g., free memory, available storage space) with a set of constants.

Note that when an app uses its own custom encryption routines, or generates random identifiers itself without relying on Android APIs, Agrigento will not be able to detect these as sources of non-determinism. However, as we explain in the next paragraph, our approach is conservative, which means this would produce a false positive, but not a false negative.

Contextualized Trace. We build the contextualized trace by incorporating the contextualized information into the raw network trace. To do this, we remove all sources of non-determinism (i.e., values stored in the contextual information) we encountered during the execution, by labeling all timestamps-related values, random identifiers, and values coming from the network, and decrypting encrypted content by mapping the recorded ciphertext values back to their plaintext. Essentially, we look at the raw network trace and try to determine, based on string comparison, values in the HTTP traffic that come from potential sources of non-determinism. This is similar to the techniques that previous works use to find certain values of private information in the network traffic. The key difference is that we do not perform the string matching to find leaks, but, rather, to *explain* sources of non-determinism. This is essentially

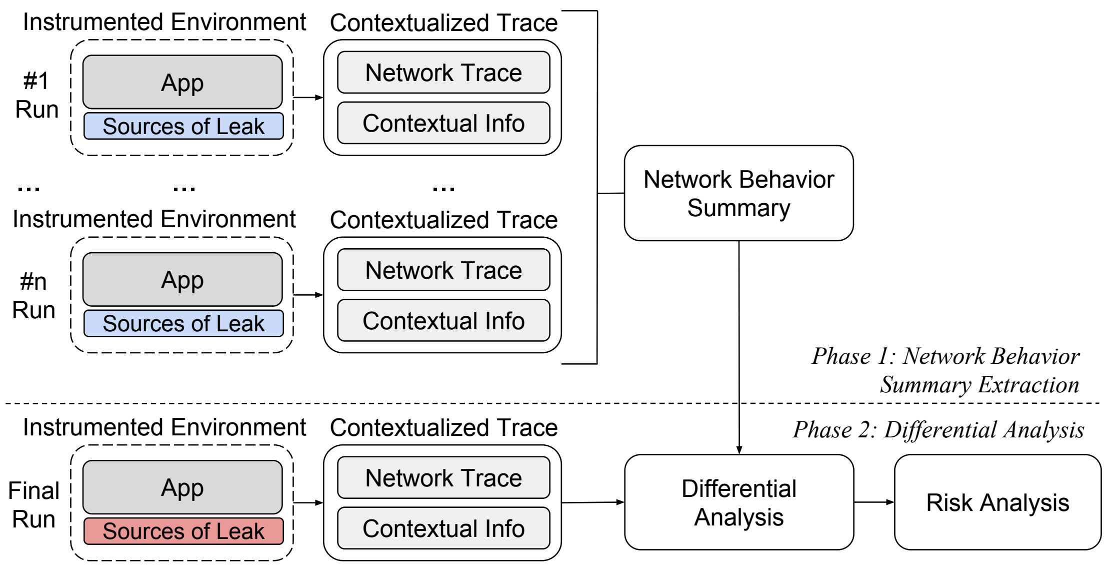
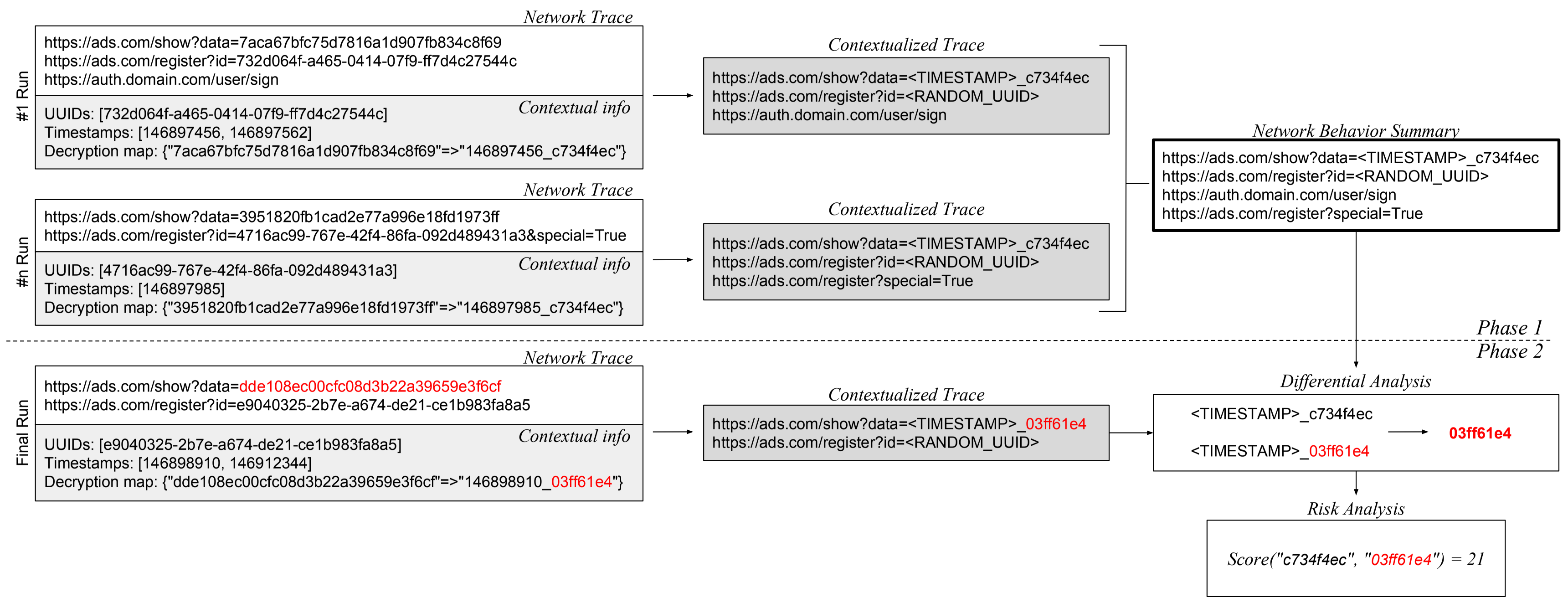


Figure 4.2: High-level overview of AGRIGENTO: during the *network behavior summary extraction* it first generates a baseline of an app's network behavior during n runs, taking into account non-determinism in the contextual information; during the *differential analysis* it then modifies the sources of private information and identifies privacy leaks based on differences in the network behavior of the final run compared to the network behavior observed in the previous runs.

the opposite goal of previous work: rather than finding leaks, we use string matching techniques to flag potential leaks as "safe." This approach has the advantage of being conservative. In fact, we flag any source of non-determinism that we cannot explain. While in previous work a failure of the string matching would lead to a missed leak (i.e., a false negative), our approach would produce, in the worst case, a false positive.

Network Behavior Summary. When AGRIGENTO builds the contextualized network traces, it essentially removes all common sources of non-determinism from the network traffic. However, it cannot fully eliminate non-determinism in the execution path of the app. Even though AGRIGENTO runs the app in an instrumented environment and replays the same sequence of events for each run, different loading times of the UI and other factors can result in different execution paths. To mitigate this issue, we run each app multiple times and merge the contextualized traces collected in the individual runs to a network behavior summary. Intuitively, the network behavior summary includes all the slightly different execution paths, generating a more complete picture of the app's network behavior. In other words, the network behavior summary represents "everything we have seen" during the executions of the given app and aims at providing a trusted baseline behavior of the app.

A distinctive aspect of AGRIGENTO is how it determines the number of times each app should be executed. Intuitively, the number of runs affects the performance of our tool in terms of false positives. However, we observed that this parameter strongly depends on the complexity of the app. Therefore, our approach is iterative and decides after each run if another one is required. After each run AGRIGENTO performs the differential analysis using the collected contextualized traces. By analyzing the discrepancies in the network behavior *without* having altered any source of private information, we can understand when AGRIGENTO has sufficiently explored the app's behavior, i.e., when the network behavior summary reaches *convergence*.



```
#1 Run
https://ads.com/show?data=7aca67bfc75d7816a1d907fb834c8f69
https://ads.com/register?id=732d064f-a465-0414-07f9-ff7d4c27544c
https://auth.domain.com/user/sign
UUIDs: [732d064f-a465-0414-07f9-ff7d4c27544c]
Timestamps: [146897456, 146897562]
Decryption map: {"7aca67bfc75d7816a1d907fb834c8f69"=>"146897456_c734f4ec"}

#n Run
https://ads.com/show?data=3951820fb1cad2e77a996e18fd1973ff
https://ads.com/register?id=4716ac99-767e-42f4-86fa-092d489431a3&special=True
UUIDs: [4716ac99-767e-42f4-86fa-092d489431a3]
Timestamps: [146897985]
Decryption map: {"3951820fb1cad2e77a996e18fd1973ff"=>"146897985_c734f4ec"}

Final Run
https://ads.com/show?data=dde108ec00cfc08d3b22a39659e3f6cf
https://ads.com/register?id=e9040325-2b7e-a674-de21-ce1b983fa8a5
UUIDs: [e9040325-2b7e-a674-de21-ce1b983fa8a5]
Timestamps: [146898910, 146912344]
Decryption map: {"dde108ec00cfc08d3b22a39659e3f6cf"=>"146898910_03ff61e4"}

Contextualized Trace (#1 Run)
https://ads.com/show?data=<TIMESTAMP>_c734f4ec
https://ads.com/register?id=<RANDOM_UUID>
https://auth.domain.com/user/sign

Contextualized Trace (#n Run)
https://ads.com/show?data=<TIMESTAMP>_c734f4ec
https://ads.com/register?id=<RANDOM_UUID>
https://ads.com/register?special=True

Contextualized Trace (Final Run)
https://ads.com/show?data=<TIMESTAMP>_03ff61e4
https://ads.com/register?id=<RANDOM_UUID>

Network Behavior Summary
https://ads.com/show?data=<TIMESTAMP>_c734f4ec
https://ads.com/register?id=<RANDOM_UUID>
https://auth.domain.com/user/sign
https://ads.com/register?special=True

Differential Analysis
<TIMESTAMP>_c734f4ec
<TIMESTAMP>_03ff61e4   →   03ff61e4

Risk Analysis
Score("c734f4ec", "03ff61e4") = 21
```

Figure 4.3: Example of how AGRIGENTO performs its analysis in two phases. (1) In the first phase it builds a network behavior summary and replaces common sources of non-determinism. (2) In the second phase performs differential analysis by changing the value of an input source of private information to identify differences in the network behavior, which it then scores as potential privacy leaks.

In practice, we say that an app reaches convergence when we do not see any discrepancies in the network behavior summary for K consecutive runs. In §4.5.3, we show how this parameter sets a trade-off between the ability of explaining non-determinism and the overall time it takes Agrigento to analyze an app (i.e., the average number of runs). Also, because some apps might never reach convergence, we set a maximum number of runs.

4.3.2 Differential Analysis

In a second phase, we run the app in the same environment as before, but modify the value of private information sources, such as the IMEI and location, we want to track. We can do this (a) once for all values to detect if an app is stealthily leaking information in general, or (b) multiple times—once for every unique identifier—to precisely identify the exact type of information the app is leaking. In the example in Figure 4.3, Agrigento changed the value of a source of private information from `c734f4ec` to `03ff61e4`.

Differential Analysis. As in the previous phase, we collect a network trace and contextual information to build a contextualized trace. Then, we compare this contextualized trace against the network behavior summary, which we extracted in the previous phase. To extract the differences, we leverage the Needleman-Wunsch algorithm [65] to perform a pairwise string sequence alignment. The algorithm is based on dynamic programming and can achieve an optimal global matching. It is well-suited for our scenario: in fact, it has been successfully applied to automatic network protocol reverse engineering efforts [12, 95, 107], which conceptually have a similar goal than our network behavior summary, in that they extract a protocol from observing the network behavior during multiple executions.

At this point of our analysis, we eliminate the final source of non-determinism: values that come from the network. For each difference, Agrigento checks if its value has been received in a response to a previous network request (e.g., the value is a server-assigned identifier). We assume that leaked information is not part of the payload of previous responses. This is reasonable since, in our threat model, the attacker does not know the value of the leaked source of private information in advance.

After this filtering step, Agrigento raises an alert for each remaining difference between the contextualized trace in the final run and the network behavior summary. This is a conservative approach, which means that, if there is some source of non-determinism Agrigento does not properly handle (e.g., apps that create UUIDs themselves or perform custom encryption without leveraging the Android framework), it will flag the app: In the worst case, this will produce a false positive.

Risk Analysis. In the last phase of our approach, Agrigento quantifies the amount of information in each identified difference to evaluate the risk that an alert is caused by an actual information leak. Our key intuition is that not all identified differences bear the same risk. Thus, we assign a score to each alert based on how much the information differs from the network behavior summary. Specifically, we leverage two distance metrics, the Hamming distance and the Levenshtein distance, to compare each alert value to the corresponding value in the network behavior summary. Finally, for each app we compute a cumulative score S as the sum of the scores of all the alerts that Agrigento produced for the app. This score provides a measure of the amount of information (i.e., the number of bits) an app can potentially leak, and it can thus be used as an indirect measure of the overall risk of a privacy leak in a given app.

4.4 System Details

We implemented Agrigento in two main components: an on-device component, which instruments the environment and collects contextual information, and the core off-device component, which intercepts the network traffic, extracts the network behavior summary, and performs the differential analysis.

4.4.1 Apps Environment Instrumentation

We implemented a module, based on Xposed [6], which hooks method calls and records and modifies their return values. As a performance optimization, Agrigento applies the contextualization steps only when needed (i.e., only when it needs to address values from a non-deterministic source).

Random values. To record Android random identifiers (UUIDs) the module intercepts the return of the Android API `randomUUID()` and reads the return value. However, recording the randomly generated values is not enough: apps frequently process these numbers (e.g., multiply them with a constant), and thus they usually do not appear directly in the network traffic. To handle this scenario, we set a fixed seed for random number generation functions. By doing so, we can observe the same values in the output network traffic for each run, even without knowing how they are transformed by an app. However, always returning the same number is also not an option since this might break app functionality. Thus, we rely on a precomputed list of randomly generated numbers. For each run, the module modifies the return value of such functions using the numbers from this precomputed list. In case the invoked function imposes constraints on the generated number (e.g., integers in the interval between 2 and 10), we adapt the precomputed numbers in a deterministic way (e.g., by adding a constant), to satisfy the specific requirements of a function call.

Timing values. Also in the case of timing information, only recording the values is not enough since timestamps are often used to produce more complex values (e.g., generation of signatures). To deal with timestamp-related values, the module hooks all the methods providing time-related information, e.g., `System.currentTimeMillis()`, stores the return values in a file during the first run, and modifies the return values reading from the file in the next runs. It reads the stored timestamps in the same order as they were written and, in case one of the next runs performs more calls to a specific method than the first run (this could be due to a different execution path), it leaves the original return values unmodified for the exceeding calls.

System values. We set to constants the return values of Android APIs that apps use to perform performance measurements and fingerprint the device for example by reading information about the available storage space from `StatFs.getAvailableBlocks()`, or by querying `ActivityManager.getMemoryInfo()` for information about available memory.

Encrypted values. In order to decrypt encrypted content, we hook the Android Crypto APIs (i.e., `Cipher`, `MessageDigest`, `Mac`) and store the arguments and return value of each method. Our module parses the API traces to build a decryption map that

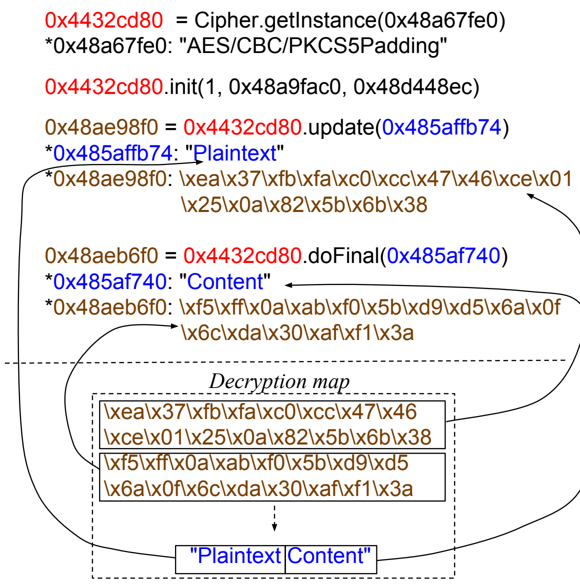

0x4432cd80 = Cipher.getInstance(0x48a67fe0)
*0x48a67fe0: "AES/CBC/PKCS5Padding"
0x4432cd80.init(1, 0x48a9fac0, 0x48d448ec)
0x48ae98f0 = 0x4432cd80.update(0x485affb74)
*0x485affb74: "Plaintext"
*0x48ae98f0: \xea\x37\xfb\xfa\xc0\xcc\x47\x46\xce\x01 \x25\x0a\x82\x5b\x6b\x38
0x48aeb6f0 = 0x4432cd80.doFinal(0x485af740)
*0x485af740: "Content"
*0x48aeb6f0: \xf5\xff\x0a\xab\xf0\x5b\xd9\xd5\x6a\x0f \x6c\xda\x30\xaf\xf1\x3a
Decryption map
\xea\x37\xfb\xfa\xc0\xcc\x47\x46 \xce\x01\x25\x0a\x82\x5b\x6b\x38
\xf5\xff\x0a\xab\xf0\x5b\xd9\xd5 \x6a\x0f\x6c\xda\x30\xaf\xf1\x3a
"Plaintext|Content"

Figure 4.4: Example of how AGRIGENTO leverages Crypto API traces to build an entry of the decryption map that maps ciphertext to its corresponding plaintext (address* represents the content stored at that *address*).**

allows it to map ciphertext to the corresponding original plaintext. Since the final ciphertext can be the result of many Crypto API calls, AGRIGENTO combines the values tracking the temporal data dependency. Figure 4.4 shows an example of how we use Crypto API traces to create a map between encrypted and decrypted content. Specifically, the example shows how AGRIGENTO creates an entry in the decryption map by tracing the API calls to a `Cipher` object and by concatenating the arguments of such calls (`update()`, `doFinal()`).

Patching JavaScript code. We observed many applications and ad libraries downloading and executing JavaScript (JS) code. Often, this code uses random number generation, time-related, and performance-related functions. We implemented a module in the proxy that inspects the JS code and patches it to remove non-determinism. Specifically, this module injects a custom random number generation function that uses a fixed seed, and replaces calls to `Math.random()` and `getRandomValues()` with our custom generator. Also, the JS injector replaces calls to time-related functions (e.g., `Date.now()`) with calls to a custom, injected timestamp generator, and sets constant values in global performance structures such as `timing.domLoading`.

4.4.2 Network Setup

Our implementation of AGRIGENTO captures the HTTP traffic and inspects GET and POST requests using a proxy based on the `mitmproxy` library [4]. In order to intercept HTTPS traffic, we installed a CA certificate on the instrumented device. Furthermore, to be able to capture traffic also in the case apps use certificate pinning, we installed `JustTrustMe` [3] on the client device, which is an Xposed module that disables certificate checking by patching the Android APIs used to perform the check (e.g., `getTrustManagers()`). However, if an app performs the certificate check using custom functionality or native code, we cannot intercept the traffic.

We limit our study to HTTP(S) traffic (further referred to both as HTTP), since

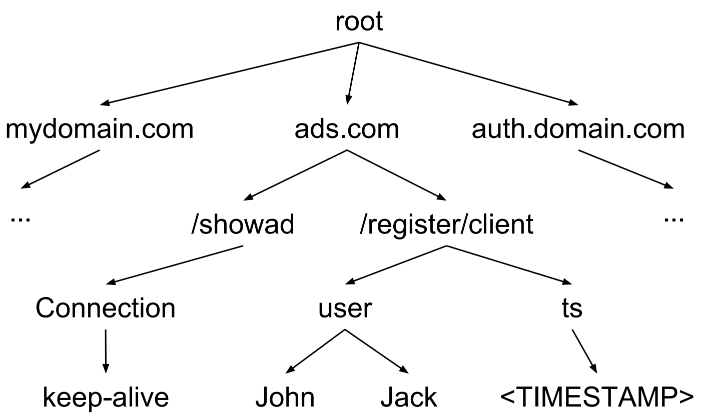


Figure 4.5: Example of the tree-based data structure used to model a network behavior summary.

related work found this to be the protocol of choice for the majority (77.77%) of Android apps [27]. However, this is only a limitation of our prototype implementation of Agrigento, and not a fundamental limitation of our approach.

Finally, to filter only the network traffic generated by the analyzed app, we use `iptables` to mark packets generated by the UID of the app, and route only those packets to our proxy.

4.4.3 Network Behavior Summary

We model the network behavior summary using a tree-based data structure, which contains the HTTP GET and POST flows from all the contextualized traces. The tree has four layers. The first layer contains all the domain names of the HTTP flows. The second layer contains the paths of the HTTP flows recorded for each domain. The third and fourth layers contain key-value pairs from the HTTP queries and HTTP headers. Also, we parse known data structures (e.g., JSON) according to the HTTP Content-Type (e.g., `application/json`). Figure 4.5 shows an example of a tree modeling a network behavior summary.

This structure is useful to group the fields of the HTTP flows that we track according to their "type" and position in the tree. In fact, when performing the differential analysis, we want to compare fields in the same position in the tree. For instance, if an HTTP request contains an HTTP value that is not part of the tree, we compare it with the other values from requests with the same domain, path, and key.

Agrigento looks for privacy leaks at all levels of the tree, i.e., in all parts of the HTTP request: the domain, path, key, and values, as well as the headers and the payload. In the current implementation Agrigento includes parsers for `application/x-www-form-urlencoded`, `application/json`, and any content that matches a HTTP query format (i.e., variable=value). However, Agrigento can be easily extended with parsers for further content types.

4.4.4 Modifying Sources of Private Information

In our implementation we track the following sources of private information: Android ID, contacts, ICCID, IMEI, IMSI, location, MAC address, and phone number. For ICCID, IMEI, IMSI, MAC address and phone number we leverage the Xposed module

Algorithm 2 Differential Analysis.

```
procedure DIFFERENTIALANALYSIS($context\_trace$, $summary$)
    diffs ←
    for $http\_flow \in context\_trace$ do
        if $http\_flow \notin summary$ then
            $field$ ← getMissingField($http\_flow, summary$)
            $fields$ ← getSamePositionField($field, summary$)
            diffs.$add$(COMPARE($field, fields$))

    return diffs

procedure COMPARE($field$, $fields$)
    diffs ←
    $most\_similar$ ← mostSimilar($field, fields$)
    if $isKnownDataStructure(field, most\_similar)$ then
        $subfields$ ← parseDataStructure($field$)
        $similar\_subfields$ ← parseDataStructure($most\_similar$)
        for $i \in subfields$ do
            diffs.$add$(COMPARE($subfields_i, similar\_subfields_i$))

        return diffs
    if $isKnownEncoding(field, most\_similar)$ then
        $field$ ← decode($field$)
        $most\_similar$ ← decode($most\_similar$)
    $alignment$ ← align($field, most\_similar$)
    $regex$ ← getRegex($alignment$)
    diffs ← getRegexMatches($field, regex$)
    diffs ← removeNetworkValues(diffs)
    diffs ← whitelistBenignLibaries(diffs)

    return diffs
```

to alter the return values of the Android APIs that allow to retrieve such data (e.g., `TelephonyManager.getDeviceId()`). For the Android ID we directly modify the value in the database in which it is stored, while to alter the contact list we generate intents through adb. We also use mock locations, which allow to set a fake position of the device for debug purposes.

4.4.5 Differential Analysis

In the second phase of our approach, AGRIGENTO modifies the input sources of private information as described in the previous section, reruns the app in the instrumented environment, and compares the new contextualized trace with the network behavior summary to identify changes in the network traffic caused by the input manipulation.

We implemented the differential analysis following the steps defined in Algorithm 2. For each HTTP flow in the contextualized trace collected from the final run, AGRIGENTO navigates the tree and checks if each field of the given flow is part of the tree. If it does not find an exact match, AGRIGENTO compares the new field with the fields in the same position in the tree (e.g., requests to the same domain, path, and key). Specifically, AGRIGENTO performs the comparison between the new field and the most similar field among those in the same position in the tree. During the comparison phase, Agrigento recognizes patterns of known data structures such as JSON. If any are found, Agrigento parses them and performs the comparison on each subfield. This step is useful to improve the alignment quality and it also improves the performance since aligning shorter subfields is faster than aligning long values. Furthermore, before the comparison, Agrigento decodes known encodings (i.e., Base64, URLencode). Then, Agrigento leverages the Needleman-Wunsch algorithm to obtain an alignment of the fields under comparison. The alignment identifies regions of similarity between the two fields and inserts gaps so that identical characters are placed in the same positions. From the alignment, Agrigento generates a regular expression. Essentially, it merges consecutive gaps, and replaces them with a wildcard (i.e., `*`). Finally, it obtains a set of differences by extracting the substrings that match the wildcards of the regular expression from a field. Agrigento then discards any differences caused by values that have been received by previous network requests (e.g., server-assigned identifier). Finally, Agrigento also whitelists benign differences caused by known Google libraries (e.g., `googleads`), which can be particularly complex to analyze and that contain non-determinism Agrigento cannot efficiently eliminate.

Example. For instance, in this simplified case, the network behavior summary tree contains the following HTTP flows:

```
domain.com/path?key=111111111_4716ac99767e
domain.com/path?key=111111111_6fa092d4891a
other.com/new?id=28361816686630788
```

The HTTP flow in the contextualized trace collected from the final run is:

```
domain.com/path?key=999999999_4716ac99767e
```

Agrigento navigates the tree from `domain.com` to `key`, and then determines that `999999999_4716ac99767e` is not part of the tree. Hence, it selects the most similar field in the tree, and performs the comparison with its value. In this case, it aligns `999999999_4716ac99767e` with `111111111_4716ac99767e`. Starting from the alignments it produces the regular expression `*_4716ac99767e` and determines `999999999` as the difference in the network behavior of the final run compared to the network behavior summary of previous runs.

4.4.6 Risk Analysis

As mentioned in §4.3.2 we combine the Hamming and the Levenshtein distance to compute a score for each of the differences Agrigento identifies during differential analysis. In particular, we are interested in quantifying the number of bits that differ in the network traffic of the final run from what we have observed in the network behavior summary.

For each field that the differential analysis flagged as being different from the previously observed network traffic, we compute a score based on the distance of its value to the most similar value in the same position of the network behavior summary. This is equivalent to selecting the minimum distance between the value and all other previously observed values for a specific field.

Given an app A, D (= the differences detected by analyzing A), and F (= all the fields in the tree of A's network behavior), we then compute an overall score S_A that quantifies how many bits the app is leaking:

$$distance(x,y) = \begin{cases} \mathsf{Hamming}(x,y) & \text{if len(x)=len(y)} \\ \mathsf{Levenshtein}(x,y) * 8 & \text{otherwise} \end{cases}$$

$$S_A = \sum_{\forall d \in D} \min_{\forall f \in F} distance(d,f)$$

We combine the Hamming and the Levenshtein distance as follows: if the values under comparison are of equal length we use the Hamming distance, otherwise we use the Levenshtein distance. While we apply the Hamming distance at the bit level, the Levenshtein distance calculates the minimum number of single-character edits. In the latter case, to obtain the number of different bits, we simply map one character to bits by multiplying it with 8. We note that this distance metric does not provide a precise measurement, but we believe it provides a useful estimation of the amount of information *contained* in each difference. Moreover, we note that BayesDroid [91] also applied the Hamming and Levenshtein distances, although only on strings of the same length, to provide a rough indication on how much information is contained in a given leak. Both metrics share the very same intuition and, therefore, provide a similar numeric result.

4.5 Experimental Results

For our evaluation, we first performed an experiment to characterize non-determinism in network traffic and demonstrate the importance of leveraging contextual information when applying differential analysis to the network traffic of mobile apps. Second, we compared the results of our technique with existing tools showing that Agrigento outperformed all of them, and identified leaks in several apps that no other tool was able to detect. Then, we describe the results of our analysis on current popular apps and present some interesting case studies describing the stealthy mechanisms apps use to leak private information. Finally, we assess the performance of Agrigento in terms of runtime.

4.5.1 Experiment Setup

We performed our experiments on six Nexus 5 phones, running Android 4.4.4, while we deployed Agrigento on a 24 GB RAM, 8-core machine, running Ubuntu 16.04. The devices and the machine running Agrigento were connected to the same subnet, allowing Agrigento to capture the generated network traffic.

We chose to perform our experiments on real devices since emulators can be easily fingerprinted by apps and ad libraries [68, 92]. Especially ad libraries are likely to perform emulator detection as ad networks, such as Google's AdMob [42], encourage the use of test ads for automated testing to avoid inflating ad impressions. By using real devices instead of emulators our evaluation is thus more realistic. Furthermore, we set up a Google account on each phone to allow apps to access the Google Play Store and other Google services.

For each execution, we run an app for 10 minutes using Monkey [1] for UI stimulation. We provide Monkey with a fixed seed so that its interactions are the same across runs. Although the fixed seed is not enough to remove all randomness from the UI interactions, it helps to eliminate most of it. At the end of each run, we uninstall the app and delete all of its data.

4.5.2 Datasets

We crawled the 100 most popular free apps across all the categories from the Google Play Store in June 2016. Additionally, we randomly selected and downloaded 100 less popular apps. We distinguish between those two datasets based on the intuition that these two sets of apps might differ significantly in their characteristics and overall complexity.

In order to compare our approach with existing techniques, we also obtained the dataset from the authors of ReCon [74], which they used to compare their approach to state-of-the-art static and dynamic data flow techniques. This dataset contains the 100 most popular free apps from the Google Play Store in August 2015 and the 1,000 most popular apps from the alternative Android market AppsApk.com. Ultimately, we use 750 of those apps for analysis, since those apps were the ones that produced any network traffic in ReCon's experiments. We further obtained the dataset of Bayes-Droid [91], which contains 54 of the most popular apps from the Google Play Store in 2013.

4.5.3 Characterizing Non-Determinism in Network Traffic

One key aspect of our work is being able to characterize and explain non-determinism in network traffic. In fact, we want to distinguish what changes "no matter what" and what changes "exactly because we modified the input." First, we show that trivially applying approaches based on differential analysis is ineffective when applied to modern Android apps. Second, our technique allows us to pinpoint which apps are problematic, i.e., for which apps we cannot determine why the network output changes. In this case, we cannot reliably correlate the differences in output with the differences in input and, therefore, we flag them as potentially leaking private information. We note that we can adopt this conservative aggressive policy only because we rarely encounter inexplicable differences in the network traffic of apps that do *not* leak private information. In other words, changes in network traffic that cannot be explained by our system are strong indicators that private information is leaked.

To demonstrate how poorly a naïve differential analysis approach without considering any network-based non-determinism would perform, we analyzed the 100 popular Google Play apps from the ReCon dataset twice: the first time, we trivially applied the differential analysis *without* leveraging any contextual information; the second time, instead, we applied our full approach, executing the apps in our instrumented environment and exploiting the collected contextual information. In both cases, we measured the number of runs needed to converge, setting 20 as the maximum number of runs.

Figure 4.6 shows the cumulative distribution functions of the number of runs required to reach convergence in the two scenarios. While in the first case almost all the apps did not reach convergence (within a maximum number of 20 runs), our approach correctly handled most of the cases. This demonstrates two things: (1) network traffic is very often non-deterministic, (2) in most cases, the contextual information recorded during the app's analysis is enough to determine the real source of non-determinism.

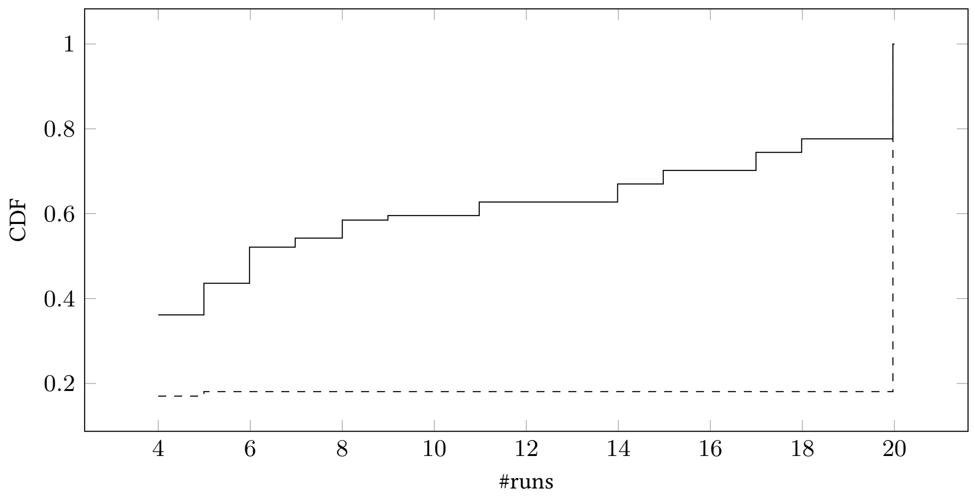


Figure 4.6: Cumulative distribution function (CDF) of the number of runs required for convergence (for $K = 3$) applying Agrigento's full approach (solid line), and the trivial differential analysis approach (dashed line) that does not consider any non-determinism in the network behavior.

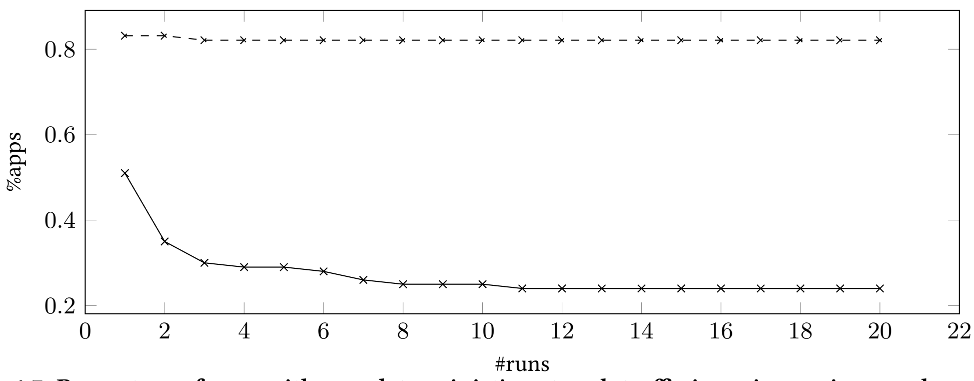


Figure 4.7: Percentage of apps with non-deterministic network traffic in an increasing number of runs when applying Agrigento's full approach (solid line), and the trivial differential analysis approach without leveraging contextual information (dashed line).

In order to further confirm this finding, we evaluated how the number of runs per app affects the number of apps for which Agrigento cannot completely explain some source of non-determinism. To do so, we performed a final execution *without* altering any source of private information, and measured the number of apps that contained non-determinism in the network traffic (i.e., the number of apps for which Agrigento raised an alert). Figure 4.7 shows that, in contrast to our full approach, when applying the differential analysis trivially, increasing the number of runs is not enough to reduce non-determinism (82.1% of the apps generated non-deterministic network traffic).

Finally, we evaluated how the choice of K (i.e., the number of consecutive runs without discrepancies considered to reach convergence) affects Agrigento's ability to explain non-determinism. We performed the evaluation on two datasets: the 100 most popular apps from the Google Play Store and 100 randomly selected less popular apps from the Google Play Store. We run the analysis *without* altering any source of private information. By doing this, any alert is caused by the fact that there is some non-determinism in the network traffic that Agrigento could not explain. Table 4.1 shows that $K = 3$ minimizes the number of apps with unexplained non-determinism in their network traffic, at the cost of a small increase in the average number of runs required per app. This evaluation also shows that the popular apps indeed seem to be more complex than the randomly selected ones, for which AGRIGENTO required a lower number of runs on average and could fully explain all sources of non-determinism in more cases overall.

4.5.4 Comparison with Existing Tools

To evaluate our approach and establish the presence of false positives and false negatives, we compared AGRIGENTO to existing state-of-the-art analysis tools. Generally, comparing the results of this kind of systems is far from trivial, the main problem being the absence of ground truth. Also, especially in the case of obfuscated leaks, the detected information leaks are often hard to verify by looking at the network traffic alone. Therefore, we manually reverse engineered the apps to the best of our ability to confirm our results. Finally, dynamic analysis results are influenced by limited coverage and different UI exploration techniques, which impedes the comparison.

The only currently available benchmark for privacy leak detection is the DroidBench[1] test suite, which is commonly used to evaluate approaches based on both static and dynamic analysis. We found, however, that it contains very few test cases for dynamic analysis, and those focus mainly on emulator detection (not affecting us since we run our evaluation on real devices). It also does not address complex obfuscation scenarios such as the ones we observed in this work, and, thus, none of the test cases are appropriate for the evaluation of AGRIGENTO.

We thus performed the comparison against existing tools using two datasets on which related work was evaluated: 750 apps from ReCon, and 54 apps from BayesDroid.

ReCon dataset. A similar comparison to evaluate state-of-the-art analysis tools from different categories (static taint analysis, dynamic taint analysis, and a combination of both) has been performed recently to evaluate ReCon [74], which itself is based on network flow analysis. Table 4.2 shows the comparison between our tool and AppAudit [101], Andrubis [61] (which internally uses TaintDroid [29]), FlowDroid [11], and ReCon. We base our comparison on the number of apps flagged by each tool for leaking information. For the comparison we considered the following sources of private information: Android ID, IMEI, MAC address, IMSI, ICCID, location, phone number, and contacts.

[1] https://github.com/secure-software-engineering/DroidBench

Table 4.1: Choice of K (= number of consecutive runs to reach convergence) and its effect on the average number of runs per app, and number of apps with non-determinism in the network traffic that AGRIGENTO cannot explain.

K	Popular		Non-Popular		All	
	#apps	avg #runs	#apps	avg #runs	#apps	avg #runs
1	39	6.02	16	3.10	55	4.56
2	30	8.28	14	4.44	44	6.36
3	**28**	**9.85**	**11**	**5.67**	**39**	**7.76**
4	28	12.42	11	6.78	39	9.60
5	28	13.82	11	8.01	39	10.92

Compared to ReCon, Agrigento detected 165 apps that ReCon did not identify, while it did not flag 42 apps that ReCon identified. We manually checked the results to verify the correctness of our approach. Among the 42 Agrigento did not detect, 23 did not generate any network traffic during our analysis. This may be due to different reasons, for instance different UI exploration (ReCon manually explored part of the dataset), or because the version of the app under analysis does not properly work in our test environment. We manually inspected the network traffic generated by the remaining 19 apps. In particular, we manually verified whether each network trace contained any of the values of the sources of private information that we considered, and we also checked for known transformations, such as MD5 hashes and Base64 encoding. In all cases, we did not identify any leak (i.e., we did not identify any false negatives). We acknowledge that this manual evaluation does not exclude the presence of false negatives. However, we consider this an encouraging result nonetheless.

To perform a more thorough evaluation of false negatives, we also performed an additional experiment. Since one main challenge when comparing approaches based on dynamic analysis is related to GUI exploration differences, we asked the authors of ReCon to run their tool on the network traffic dumps we collected during our analysis. In this way, it is possible to compare both tools, ReCon and Agrigento, on the same dynamic trace. On this dataset, ReCon flagged 229 apps for leaking information. Agrigento correctly detected all the apps identified by ReCon, and, in addition, it detected 49 apps that ReCon did not flag. This evaluation shows that, also for this experiment, Agrigento did not show any false negatives. Moreover, we also looked for false positives, and we manually verified the 49 apps detected by Agrigento and not by ReCon. Our manual analysis revealed that 32 of the 49 apps did indeed leak at least one source of private information, which should then be considered as true positives (and false negatives for ReCon). For further 5 apps we could not confirm the presence of a leak and thus classify them as false positives produced by our system. We cannot classify the remaining 12 cases as either true or false positives because of the complexity of reversing these apps.

BayesDroid dataset. We obtained the dataset used by BayesDroid and analyzed the apps with Agrigento. For the comparison we considered the common sources of information supported by both Agrigento and BayesDroid (i.e., IMEI, IMSI, Android ID, location, contacts). BayesDroid flagged 15 of the 54 apps. However, since this dataset contains older app versions (from 2013) 10 apps did not work properly or did not generate any network traffic during our analysis. Nevertheless, Agrigento flagged 21 apps, including 10 of the 15 apps identified by BayesDroid. As we did for the ReCon dataset, we manually looked at the network traces of the remaining 5 apps and we did not see any leak (3 of them did not produce any network traffic, furthermore Bayes-

Table 4.2: Comparison of Agrigento with existing tools on the ReCon dataset (750 apps)

Tool (Approach)	#Apps detected
FlowDroid (Static taint analysis)	44
Andrubis/TaintDroid (Dynamic taint analysis)	72
AppAudit (Static & dynamic taint flow)	46
ReCon (Network flow analysis)	155
Agrigento	278

Table 4.3: Number of apps detected by AGRIGENTO in the 100 most popular apps (July 2016) from the Google Play Store. The column "Any" refers to the number of apps that leak at least one of the private information sources.

	Results	Any	Android ID	IMEI	MAC Addr	IMSI	ICCID	Location	Phone Num	Contacts
TPs	Plaintext	31	30	13	5	1	0	1	0	0
	Encrypted	22	18	9	3	5	0	0	0	0
	Obfuscated	11	8	5	6	0	0	1	0	0
	Total	42	38	22	11	6	0	1	0	0
FPs		4	5	9	11	13	13	11	16	13

Droid used manual exploration of all apps). Interestingly, AGRIGENTO detected 11 apps that BayesDroid did not. We found that 6 of these apps used obfuscations that Bayes-Droid does not detect. For instance, one app included the InMobi SDK that performs a series of encodings and encryptions on the Android ID before leaking it. We describe this case in detail in §4.5.6. Moreover, the other 5 apps used Android APIs to hash or encrypt data structures (e.g., in JSON format) containing private information sources, again showing that our system detects cases that previous work cannot.

4.5.5 Privacy Leaks in Popular Apps

To evaluate AGRIGENTO on a more recent dataset, we analyzed the current (July 2016) 100 most popular apps from the Google Play Store in more detail. AGRIGENTO identified privacy leaks in 46 of the 100 apps. We manually verified the results of our analysis to measure false positives. We found that 42 of these apps are true positives, that is, they leak private information, while four apps were likely false positives. Note that, in some cases, to distinguish true positives from false positives we had to manually reverse the app. During our manual analysis, we did not encounter any false negative. Once again, we acknowledge that, due to the absence of a ground truth, it is not possible to fully exclude the presence of false negatives. In particular, as further discussed in §4.6, AGRIGENTO is affected by a number of limitations, which a malicious app could take advantage of.

We then used our risk analysis to rank the risk associated with these false positives. Interestingly, we found that while two of the four apps that caused false positives have high scores (i.e., 8,527 and 8,677 bits), for the other two apps, one in particular, AGRIGENTO assigned low scores of 6 and 24 bits. We note that although for this work we use our risk analysis only to rank the risk of a data leak in each detected app, we believe it could be used to build, on top of it, a further filtering layer that discards low bandwidth leaks. We will explore this direction in future work.

We further classified the type of leak in three groups: plaintext, encrypted, and obfuscated. The first group contains apps that leak the information in plaintext. The second group contains apps for which we observed the leaked information only after our decryption phase (i.e., the leaked value has been encrypted or hashed using the Android APIs). Finally, the third group contains apps that obfuscate information leaks by other means (i.e., there is no observable evidence of the leaked value in the network traffic).

As a first experiment, we considered leaks only at the app level since we are interested in determining whether an app leaks information or not, independently from the number of times. In other words, we are interested to determine whether a given app leaks any sensitive information. Thus, for each app analysis we performed just one final run for which we modified all the sources simultaneously. As a result, Agrigento produces a boolean output that indicates whether an app leaks private information or not, without pointing out which particular source has been leaked. Table 4.3 shows the results of this experiment. For this experiment, we consider an app as a true positive when it leaks any of the monitored sources and Agrigento flags it, and as a false positive when Agrigento flags it although it does not leak any information.

While this experiment provides valuable insights, it provides only very coarse-grained information. Thus, as a second experiment, we performed the same evaluation but we looked at each different source of information individually. In this case, we ran the app and performed the differential analysis changing only one source at a time, and we consider an app as a true positive only if it leaks information from the *modified* source and Agrigento correctly identifies the leak. Our evaluation shows that, while Agrigento produces higher false positives in identifying leaks for a specific source of information, it has very few false positives in detecting privacy leaks in general. The higher false positive rate is due to some sources of non-determinism that Agrigento failed to properly handle and that consequently cause false positives when an app does not leak data. For instance, consider the scenario in which an app leaks the Android ID and also contains some non-determinism in its network traffic that Agrigento could not eliminate. In this case, when considering leaks at app-level granularity, we consider the app as a true positive for the Android ID, since it does leak the Android ID. Instead, for any other source of information (e.g., the phone number) we consider the app as a false positive because of the non-determinism in the network traffic. Finally, we could not classify 9 apps, for which Agrigento identified leaks of some of the sources, because of the complexity of reversing these apps.

4.5.6 Case Studies

We manually reversed some apps that Agrigento *automatically* identified as leaking obfuscated or encrypted information. Here, we present some case studies showing that current apps use sophisticated obfuscation and encryption techniques. Hence, as confirmed by the results of our evaluation, state-of-the-art solutions to identify privacy leaks are not enough since they do not handle these scenarios and mostly only consider standard encodings.

Interestingly, all the leaks we found in these case studies were performed by third-party libraries, and thus may concern all the apps using those libraries.

Case study 1: InMobi. We found that InMobi, a popular ad library, leaks the Android ID using several layers of obfuscation techniques. The Android ID is hashed and XORed with a randomly generated key. The XORed content is then encoded using Base64 and then stored in a JSON-formatted data structure together with other values. The JSON is then encrypted using RSA (with a public key embedded in the app), encoded using Base64 and sent to a remote server (together with the XOR key). Figure 4.8 shows an example of such a request leaking the obfuscated Android ID. Agrigento automatically identified 20 apps in our entire dataset leaking information to InMobi domains, including one app in the 100 most popular apps from the Google

```
http://i.w.inmobi.com/showad.asm?u-id-map=iB7WTkCLJvNsaEQakKKXFhk8ZEIZlnL0jqbbYexc
BAXYHH4wSKyCDWVfp+q+FeLFTQV6jS2Xg97liEzDkw+XNTghe9ekNyMnjypmgiu7xBS1TcwZmFxYOjJkgP
OzkI9j2lryBaLlAJBSDkEqZeMVvcjcNkx+Ps6SaTRzBbYf8UY=&u-key-ver=2198564
---------------------------------------------------------------------------------

https://h.online-metrix.net/fp/clear.png?ja=33303426773f3a3930643667663b3338383130
3d343526613f2d363830247a3f363026663d333539347a31323838266c603d687c7672253163253066
253066616f6e74656e762f6a732c746370626f7926636f652466723f6a747670253161273266253266
616d6d2e65616f656b69726b7573267270697867636e617730266a683d656164376137323164313 53c
65613a31386e67606563303736363936343433363266d64643f6561633336303b64336a393531666330
3666636137326136 3a6161633563676126 6d66733f353b32306d383230613230643b6534643934383a
31663636623b323237676161 26616d65613d313933333133333133313133313331 2661743d6365656e
765f6f6f6a696c6d26617e3f76727771 74666566676e6665722b6d6f606b6c652733632b392e322634
2d3b
```

Figure 4.8: Example of the requests performed by InMobi and ThreatMetrix libraries. InMobi leaks the Android ID, as described in §4.5.6, in the value of `u-id-map`. ThreatMetrix leaks the Android ID, location, and MAC address in the `ja` variable.

Play Store. Indeed, according to AppBrain[2], InMobi is the fourth most popular ad library (2.85% of apps, 8.37% of installs).

Case study 2: ThreatMetrix. The analytics library ThreatMetrix leaks multiple sources of private information using obfuscation. It first puts the IMEI, location, and MAC address in a HashMap. It then XORs this HashMap with a randomly generated key, hex-encodes it, and then sends it to a remote server. Figure 4.8 shows an example of such a request leaking the obfuscated Android ID, location, and MAC address. We found 15 instances of this scenario in our entire dataset, one of which is part of the 100 most popular apps from the Google Play Store. According to AppBrain, ThreatMetrix SDK is used by 0.69% of the apps in the Google Play Store, and is included by 4.94% of the installs.

Further ad libraries. We found several other apps and ad libraries (MobileAppTracking, Tapjoy) leaking private information using the Android encryption and hashing APIs. In the most common scenario, the values are combined in a single string that is then hashed or encrypted. In this scenario, even though the app uses known encodings or cryptographic functions, previous tools are not able to detect the leak of private information.

4.5.7 Performance Evaluation

We execute each app for 10 minutes during each run. The analysis time per app mainly depends on the complexity of the app (i.e., the number of runs required to reach convergence). Setting $K = 3$, Agrigento analyzed, on average, one app in 98 minutes. Note that, while we executed each run sequentially, our approach can easily scale using multiple devices or emulators running the same app in parallel.

4.6 Limitations and Future Work

While we addressed the major challenges for performing differential analysis despite the overall non-determinism of the network traffic of mobile apps, our overall approach and the implementation of Agrigento still have some limitations.

[2] http://www.appbrain.com/stats/libraries/ad

Even though Agrigento improves over the existing state-of-the-art, it still suffers from potential false negatives. For example, as any other approach relying on the actual execution of an app, Agrigento suffers from limited code coverage, i.e., an app might not actually leak anything during the analysis, even if it would leak sensitive data when used in a real-world scenario. This could happen for two main reasons: (a) An app could detect that it is being analyzed and does not perform any data leaks. We address this issue by performing our analysis on real devices; (b) The component of the app that leaks the data is not executed during analysis, for example due to missing user input. We currently use Monkey, which only generates pseudorandom user input and cannot bypass, for example, login walls. Related works such as BayesDroid and ReCon performed manual exploration of apps at least for part of the dataset, which also included providing valid login credentials. Unfortunately, manual exploration is only feasible for small-scale experiments and not on a dataset of over one thousand apps such as ours, especially given the fact that Agrigento needs to generate the same consistent user input over multiple executions. As part of our future work, we are planning to explore whether it is possible to provide manual inputs for the first run of an app, and then replaying the same input with tools such as RERAN [40] in the subsequent runs. One option for collecting the initial manual inputs at scale is Amazon Mechanical Turk.

Second, Agrigento still suffers from some covert channels that an attacker could use to leak information without being detected. For instance, a sophisticated attacker could leak private information by encoding information in the number of times a certain request is performed. However, this scenario is highly inefficient from the attacker point of view. Furthermore, we could address this issue with a more accurate description of the “network behavior summary.” As a matter of fact, Agrigento severely limits the bandwidth of the channel an attacker can use to stealthily trasmit private data.

We need to run each app multiple times: by nature, an approach using differential analysis requires at least two executions, one with the original inputs, and another one with different inputs to observe changes in the outputs. As we discussed in our evaluation, the non-deterministic network behavior of modern apps further requires us to perform the original execution more than once to build a more accurate network behavior summary. Since we conservatively flag any changes in the output as a possible leak, in practice the number of runs is a trade-off between the overall analysis time and the false positive rate. Furthermore, we perform the final run once for each source of private information that we track. This requirement could be relaxed if our goal was to find privacy leaks in general, and not specific types of information. In our evaluation we performed all runs of a specific app consecutively on the same device. We could parallelize this process on different devices, however, with less control over device-specific artifacts that could potentially influence our analysis.

On the implementation side we suffer from two main limitations: First, we currently do not instrument calls to `/dev/random`, which could be used by native code directly as a source of randomness. We leave this issue for future work. Second, we are limited by the protocols we track: we only check HTTP GET and POST requests for leaks (and man-in-the-middle HTTPS even with certificate pinning in most cases). However, we share this limitation with other tools, such as ReCon, and leave an extension of Agrigento to other protocols for future work.

By design, Agrigento can only determine that a specific piece of private information was leaked, but not automatically determine how it was obfuscated. We can, however, perform the naïve approach employed by related tool of simple grepping for widely-used encodings and hashing algorithms of the value, to filter out those cases and focus manual reverse engineering efforts on the more complex and interesting ones.

Finally, we can only speculate why app developers are adopting the stealth techniques that we have uncovered in our analysis. This development could be related to the increasing awareness and opposition of users to the collection of their private data, as well as the investigative efforts of regulators such as the FTC. Currently, InMobi is very open about the data it collects in its privacy policy.[3] For future work we could investigate any malicious intent or deceptive practice behind sophisticated obfuscation techniques, based on automatically verifying whether those leaks are in violation of an app's privacy policy or not. Related work in this direction by Slavin [81] has so far only compared privacy policies against information flows identified with FlowDroid, but has not considered cases in which apps are hiding their leaks with the techniques Agrigento uncovered.

4.7 Related Work

Static taint analysis of Android apps is an active research topic, as several aspects of Android apps proved to be very challenging—in particular their component-based architecture and the multitude of entry points due to their user-centric nature and complex lifecycle. AndroidLeaks [39] was one of the first static taint analysis approaches, but lacks precision as it tracks data flow at the object-level instead of tainting their individual fields. FlowDroid [11] is more precise in this regard and one of the most widely used static taint analysis tools. Further approaches include EdgeMiner [19], which addresses the issue of reconstructing implicit control flow transitions, and Amandroid [94] and IccTA [59], which deal with inter-component instead of just intra-component data leaks. MorphDroid [37] argues that conventional data flow tracking approaches are too coarse-grained, and tracks atomic units of private information instead of the complete information (i.e., longitude and latitude instead of the location) to account for partial leaks. AppIntent [103] proposes to distinguish between user-intended and covert data leaks and uses symbolic execution to determine if a privacy leak is a result of user interaction. AppAudit [101] addresses the false positives of related static analysis approaches and verifies the detected leaks through approximated execution of the corresponding functions.

Dynamic taint analysis tracks information flow between sources of private information and sinks, e.g., the network, during runtime, either by modifying the device OS (TaintDroid [29]), the platform libraries (Phosphor [13]), or the app under analysis (Uranine [72]). AppFence [46] extends TaintDroid to detect obfuscated and encrypted leaks, and also performed a small-scale study on the format of leaks, but only found the ad library Flurry leaking data in non-human readable format in 2011— a situation that has drastically changed since then as we showed in our study. BayesDroid [91] is similar to TaintDroid, but addresses the problem of partial information leaks. It com-

[3] `http://www.inmobi.com/privacy-policy/`

pares tainted data tracked from a source of private information to a network sink, and uses probabilistic reasoning to classify a leak based on the similarity between the data at both points. While aforementioned approaches only track data flow in the Dalvik VM, there also exist approaches that also can track data flow in native code: DroidScope [102] and CopperDroid [90] perform full system emulation and inspect both an app's Dalvik and native code for the purpose of malware analysis, while the recent TaintART extends TaintDroid to native code [87]. However, ultimately, taint analysis approaches are vulnerable to apps deliberately disrupting the data flow: ScrubDroid [77] discusses how dynamic taint analysis systems for example can be defeated by relying on control dependencies (which related approaches usually do not track), or by writing and reading a value to and from system commands or the file system.

Most recently, related work has explored detecting privacy leaks at the network level, usually through network traffic redirection by routing a device's traffic through a virtual private network (VPN) tunnel and inspecting it for privacy leaks on the fly. Tools such as PrivacyGuard [82], AntMonitor [57], and Haystack [73], perform their analysis on-device using Android's built-in VPNService, but rely on hardcoded identifiers, or simply grep for a user's private information. Liu et al. [62] inspect network traffic at the ISP-level and identify private information leaks based on keys generated from manual analysis and regular expressions. Encryption and obfuscation are out of scope of the analysis, as the authors assume this scenario is only a concern for malware. ReCon [74] is another VPN-based approach, which uses a machine learning classifier to identify leaks and can deal with simple obfuscation. In the end, it relies on the data on which it is trained on—which can come from manual analysis and dynamic taint analysis tools—and it could benefit from a technique such as Agrigento to deal with more complex obfuscation techniques.

Information leakage is not a new problem and not unique to Android apps: related work on desktop applications has focused on identifying (accidental) leaks of private information through differential analysis at the process-level. TightLip [105] and Croft et al. [26] perform differential analysis on the output of two processes, one with access to private data, and one without. Both consider timestamp-related information and random seeds as sources of non-determinism and share them between processes. Ultimately, their main goal is to prohibit the accidental leakage of private information, more specifically, sensitive files, and not obfuscated content. To this end, TightLip checks if the system call sequences and arguments of the two processes diverge when the private input changes, and consequently raises an alarm if the output is sent to a network sink. In contrast, Croft et al. only allow the output of the process without access to private information to leave the internal company network. The approach of Privacy Oracle [48] is related to Agrigento: it identifies privacy leaks based on divergences in the network traffic when private input sources are modified. However, it mainly addresses non-determinism at the OS-level (i.e., performing deterministic executions using OS snapshots) and does not consider non-determinism in network traffic. In fact, it cannot handle random tokens in the network traffic, nor encryption, and produces false positives when messages in network flows are reordered between executions.

Finally, Shu et al. [80] propose a sequence alignment algorithm for the detection of obfuscated leaks in files and network traffic, which assigns scores based on the amount of private information they contain. While this approach focuses on the detection of obfuscated leaks, it explicitly does not address intentional or malicious leaks, and only considers character replacement, string insertion and data truncation.

In contrast to related work, we are the first to address the topic of obfuscation of privacy leaks in order to deal with adversaries, i.e., apps or ad libraries actively trying to hide the fact that they are leaking information. As we have shown in our evaluation, this is a very realistic threat scenario and a practice that is already common amongst popular mobile apps and ad libraries.

4.8 Concluding Remarks

We showed that while many different approaches have tackled the topic of privacy leak detection in mobile apps, it is still relatively easy for app and ad library developers to hide their information leaks from state-of-the-art tools using different types of encoding, formatting, and cryptographic algorithms. This chapter introduced Agrigento, a new approach that is resilient to such obfuscations and, in fact, to any arbitrary transformation performed on the private information before it is leaked. Agrigento works by performing differential black-box analysis on Android apps. We discussed that while this approach seems intuitive, in practice, we had to overcome several key challenges related to the non-determinism inherent to mobile app network traffic.

One key insight of this work is that non-determinism in network traffic can be often explained and removed. This observation allowed us to develop novel techniques to address the various sources of non-determinism and it allowed us to conservatively flag any deviations in the network traffic as potential privacy leaks. In our evaluation on 1,004 Android apps, we showed how Agrigento can detect privacy leaks that state-of-the-art approaches cannot detect, while, at the same time, only incurring in a small number of false positives. We further identified interesting cases of custom and complex obfuscation techniques, which popular ad libraries currently use to exfiltrate data without being detected by other approaches.

5. Conclusions

In this dissertation we described our effort to analyze current software abuses and propose novel approaches to protect users from such threats. We focused on banking Trojans, ransomware, and mobile privacy leaks. All these threats share the common motivation behind cybercriminals modus operandi, which is obtaining financial gains abusing users' data.

First, we studied banking Trojans, a particular kind of malware that steals banking credentials by injecting malicious code into the browser and taking control of the victim's browser session. We presented Prometheus, a platform that analyzes such Trojans and extracts robust, behavioral signatures of their malicious behavior. The produced signatures allow to check, on the client side, whether a web page is currently being rendered on an infected machine or, more in general, if a page of interest is targeted by a specific sample. We developed a prototype, Iris, which, leveraging Prometheus's signatures, automatically detects Man-in-the-Browser attacks that result in visible DOM modifications, independently from the malware implementation.

Second, we focused on ransomware, a large class of malware that encrypts users' file and asks for a ransom in order to obtain the decryption key(s) needed to recover the original files. We first studied how ransomware compares to benign software from the filesystem's viewpoint, in order to identify strong detection criteria. However, we observed that in some particular scenarios, such as the ransomware one, pure detection approaches are not enough. Instead, we proposed a proactive approach that equips modern operating systems with self-healing capabilities. Thus, if a file is surreptitiously altered by one or more malicious processes, the filesystem presents the original, mirrored copy to the user space applications. This shadowing mechanism is dynamically activated and deactivated depending on the outcome of the aforementioned detection logic. We implemented our approach in ShieldFS, an innovative tool that makes the Windows native filesystem immune to ransomware attacks by detecting malicious activities and transparently reverting the effects of such attacks. We evaluated ShieldFS against distinct ransomware families, showing that it can successfully protect user data from real-world attacks.

Third, we focused on mobile privacy issues proposing an approach to detect privacy leaks in an obfuscated-resilient fashion. In fact, while many different approaches have tackled the topic of privacy leak detection in mobile apps, app and ad library developers can easily hide their information leaks from state-of-the-art tools using different types of encoding, formatting, and cryptographic algorithms. Our approach, Agrigento, is instead based on network black-box differential analysis and it is resilient to any obfuscation technique. However, trivially applying differential analysis at the network traffic is not feasible, because of the non-determinism inherent to mobile app network traffic. One key insight of our research is that we found that this non-determinism can be often explained and removed. Hence, we proposed novel techniques to address different sources of non-determinism, making the network differential analysis practical. We evaluated Agrigento against popular Android apps and our experiments showed that there exists several libraries that employ different encryption and obfuscation techniques to hide the fact they are leaking sensitive information.

Bibliography

[1] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[2] Internet users. URL http://www.internetlivestats.com/internet-users/.

[3] JustTrustMe. https://github.com/Fuzion24/JustTrustMe.

[4] mitmproxy. https://mitmproxy.org.

[5] Video demonstration of shieldfs in action. URL https://www.youtube.com/watch?v=0UlgdnQQaLM.

[6] Xposed framework. http://repo.xposed.info.

[7] Unlock the key to repel ransomware. Technical report, Kaspersky Lab, 2015.

[8] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupe, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[9] Nicoló Andronio, Stefano Zanero, and Federico Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *Research in Attacks, Intrusions, and Defenses*, pages 382–404. Springer, 2015.

[10] Liviu Arsene and Alexandra Gheorghe. Ransomware. a victim's perspective. Technical report, Bitdefender, 2016. URL http://www.bitdefender.com/media/materials/white-papers/en/Bitdefender_Ransomware_A_Victim_Perspective.pdf.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[12] Marshall Beddoe. The Protocol Informatics Project. http://www.4tphi.net/~awalters/PI/PI.html, 2004.

[13] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014.

[14] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.

[15] Blueliv. Chasing cybercrime: network insights of dyre and dridex trojan bankers, 2015.

[16] Jean-Ian Boutin. The evolution of webinjects. In *Virus Bulletin Conference*, 2014.

[17] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Commun. ACM*, 20(10):762–772, Oct 1977. ISSN 0001-0782. doi: 10.1145/359842.359859.

[18] Armin Buescher, Felix Leder, and Thomas Siebert. Banksafe information stealer detection inside the web browser. In *Recent Advances in Intrusion Detection*, pages 262–280. Springer, 2011.

[19] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[20] Cisco. Cisco visual networking index: Forecast and methodology, 2016–2021, 2017. URL `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf`.

[21] Fred Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[22] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. ShieldFS: A self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Computer Security Applications Conference*. ACM.

[23] Andrea Continella, Michele Carminati, Mario Polino, Andrea Lanzi, Stefano Zanero, and Federico Maggi. Prometheus: Analyzing WebInject-based information stealers. *Journal of Computer Security*, 2017.

[24] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[25] Claudio Criscione, Fabio Bosatelli, Stefano Zanero, and Federico Maggi. Zarathustra: Extracting webinject signatures from banking trojans. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pages 139–148. IEEE, 2014.

[26] Jason Croft and Matthew Caesar. Towards Practical Avoidance of Information Leakage in Enterprise Networks. In *Proc. of the USENIX Conference on Hot Topics in Security (HotSec)*, 2011.

[27] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[28] Gerry Eisenhaur, Michael N. Gagnon, Tufan Demir, and Neil Daswani. Mobile Malware Madness, and How to Cap the Mad Hatters. A Preliminary Look at Mitigating Mobile Malware. In *Black Hat USA (BH-US)*, 2011.

[29] 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 *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[30] Nicolas Falliere and Eric Chien. Zeus: King of the bots. *Symantec Security Response (http://bit. ly/3VyFV1)*, 2009.

[31] Aristide Fattori, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. Hypervisor-based malware protection with accessminer. *Computers & Security*, 52:33–50, 2015.

[32] FBI. The fraud scheme, 2010. URL `http://www.fbi.gov/news/stories/2010/october/cyber-banking-fraud`.

[33] FBI. Criminals continue to defraud and extort funds from victims using cryptowall ransomware schemes, 2015. URL `http://www.ic3.gov/media/2015/150623.aspx`.

[34] Federal Trade Commission. FTC Approves Final Order Settling Charges Against Flashlight App Creator. `https://www.ftc.gov/news-events/press-releases/2014/04/ftc-approves-final-order-settling-charges-against-flashlight-app`, April 2014.

[35] Federal Trade Commission. Two App Developers Settle FTC Charges They Violated Children's Online Privacy Protection Act. `https://www.ftc.gov/news-events/press-releases/2015/12/two-app-developers-settle-ftc-charges-they-violated-childrens`, December 2015.

[36] Federal Trade Commission. Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission. `https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked`, June 2016.

[37] Pietro Ferrara, Omer Tripp, and Marco Pistoia. MorphDroid: Fine-grained Privacy Verification. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[38] Michael N. Gagnon. Hashing IMEI numbers does not protect privacy. `http://blog.dasient.com/2011/07/hashing-imei-numbers-does-not-protect.html`, 2011.

[39] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proc. of the International Conference on Trust and Trustworthy Computing (TRUST)*, 2012.

[40] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2013.

[41] Max Goncharov. Russian underground 101. *Trend Micro incorporated research paper*, page 51, 2012.

[42] Google. AdMob Behavioral Policies. `https://support.google.com/admob/answer/2753860?hl=en`, 2016.

[43] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 821–832. ACM, 2012.

[44] Shay Gueron. Intel advanced encryption standard (aes) new instructions set. Technical report, Intel, 2012. URL `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`.

[45] Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: Detection and mitigation of malicious websites with a frozen dom. In *Recent Advances in Intrusion Detection*, pages 281–300. Springer, 2011.

[46] 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 *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[47] Microsoft Inc. File system minifilter drivers, 2014. URL `https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402(v=vs.85).aspx`.

[48] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[49] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. Unveil: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 757–772, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4.

[50] Loucif Kharouni. Automating Online Banking Fraud, 2012.

[51] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148, page 3. Springer, 2015.

[52] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference.* ACM, 2011.

[53] Vadim Kotov and Mantej Singh Rajpal. Understanding crypto-ransomware: In-depth analysis of the most popular malware families. Technical report, Bromium, 2014.

[54] Brian Krebs. The equifax breach: What you should know, 2017. URL `https://krebsonsecurity.com/2017/09/the-equifax-breach-what-you-should-know/`.

[55] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412. ACM, 2010.

[56] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412. ACM, 2010.

[57] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In *Proc. of the ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data (C2BID)*, 2015.

[58] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 203–214. ACM, 2015.

[59] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2015.

[60] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 349–358. ACM, 2012.

[61] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proc. of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[62] Yabing Liu, Han Hee Song, Ignacio Bermudez, Alan Mislove, Mario Baldi, and Alok Tongaonkar. Identifying Personal Information in Internet Traffic. In *Proc. of the ACM Conference on Online Social Networks (COSN)*, 2015.

[63] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android System. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[64] Trend Micro. Ransomware bill seeks to curb the extortion malware epidemic, 2016. URL `http://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/ransomware-bill-curb-the-extortion-malware-epidemic`.

[65] Saul B Needleman and Christian D Wunsch. A General Method Applicable to Search for Similarities in Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[66] Mariani Nicola, Continella Andrea, Pogliani Marcello, Carminati Michele, Maggi Federico, and Zanero Stefano. Poster: Detecting webinjects through live memory inspection. *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[67] Thomas Ormerod. *An Analysis of a Botnet Toolkit and a Framework for a Defamation Attack.* PhD thesis, Concordia University, 2012.

[68] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proc. of the European Workshop on System Security (EuroSec)*, 2014.

[69] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications, 2014.

[70] Ashkan Rahimian, Raha Ziarati, Stere Preda, and Mourad Debbabi. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*, pages 408–425. Springer, 2014.

[71] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[72] Vaibhav Rastogi, Zhengyang Qu, Jedidiah McClurg, Yinzhi Cao, Yan Chen, Weiwu Zhu, and Wenzhi Chen. Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android. In *Proc. of the International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2015.

[73] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419*, 2015.

[74] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2016.

[75] Marco Riccardi, Roberto Di Pietro, and Jorge Aguila Vila. Taming zeus by leveraging its own crypto internals. In *eCrime Researchers Summit (eCrime), 2011*, pages 1–9. IEEE, 2011.

[76] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 65–79. IEEE, 2012.

[77] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed˜Ali Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *Proc. of the International Conference on Security and Cryptography (SECRYPT)*, 2013.

[78] Kevin Savage, Peter Coogan, and Hon Lau. The evolution of ransomware. Technical report, Symantec, 2015.

[79] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. Cryptolock (and drop it): Stopping ransomware attacks on user data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016.

[80] X. Shu, J. Zhang, D. D. Yao, and W. C. Feng. Fast Detection of Transformed Data Leaks. *IEEE Transactions on Information Forensics and Security*, 11(3):528–542, March 2016.

[81] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2016.

[82] Yihang Song and Urs Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proc. of the Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015.

[83] Aditya K Sood, Richard J Enbody, and Rohit Bansal. Dissecting spyeye–understanding the design of third generation botnets. *Computer Networks*, 57(2):436–450, 2013.

[84] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, chapter BitIodine: Extracting Intelligence from the Bitcoin Network, pages 457–468. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-45472-5. doi: 10.1007/978-3-662-45472-5_29.

[85] Abhinav Srivastava, Andrea Lanzi, Jonathon Giffin, and Davide Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 214–233, 2011.

[86] Doherty Stephen, Krysiuk Piotr, and Wueest Candid. The state of financial trojans 2013, 2013. URL `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_state_of_financial_trojans_2013.pdf`.

[87] Mingshen Sun, Tao Wei, and Lui John. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[88] Symantec. Dyre: Emerging threat on financial fraud landscape, 2015.

[89] Symantec. Internet security threat report 2017, 2017. URL `https://s1.q4cdn.com/585930769/files/doc_downloads/lifelock/ISTR22_Main-FINAL-APR24.pdf`.

[90] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[91] Omer Tripp and Julia Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *Proc. of the USENIX Security Symposium*, 2014.

[92] Timothy Vidas and Nicolas Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[93] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[94] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[95] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic Network Protocol Analysis. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2008.

[96] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 98–118. Springer, 2015.

[97] Candid Wueest. The state of financial Trojans 2014, 2014.

[98] Candid Wueest. Financial threats review 2017, 2017. URL `https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-financial-threats-review-2017-en.pdf`.

[99] James Wyke. Vawtrak - International Crimeware-as-a-Service, 2014.

[100] James Wyke. Breaking the bank(er): automated configuration data extraction for banking malware, 2015. URL `https://www.sophos.com/en-us/medialibrary/PDFs/technicalpapers/sophos-wyke-breaking-the-bank-VB2015.pdf`.

[101] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective Real-time Android Application Auditing. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[102] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. of the USENIX Security Symposium*, 2012.

[103] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[104] Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 129–140. IEEE, 1996.

[105] Aydan R Yumerefendi, Benjamin Mickle, and Landon P Cox. TightLip: Keeping Applications from Spilling the Beans. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[106] Bu Zheng, Bueno Pedro, Kashyap Rahul, and Wosotowsky Adam. The new era of botnets, 2013.

[107] Chaoshun Zuo, Wubing Wang, Rui Wang, and Zhiqiang Lin. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.