The Dissertation Committee for Mikhail Kazdagli
certifies that this is the approved version of the following dissertation:

Robust Behavioral Malware Detection

Committee:

Mohit Tiwari, Supervisor

Sanjay Shakkottai

Sarfraz Khurshid

Milos Gligoric

Mihai Christodorescu

Robust Behavioral Malware Detection

by

Mikhail Kazdagli

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2018

Robust Behavioral Malware Detection

Publication No. _____

Mikhail Kazdagli, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Mohit Tiwari

Computer security attacks evolve to evade deployed defenses. Recent attacks have ranged from exploiting generic software vulnerabilities in memory-unsafe languages such as buffer overflows and format string vulnerabilities to exploiting logic errors in web applications, through means such as SQL injection and cross-site scripting. Furthermore, recent attacks have focused on escalating privileges and stealing sensitive information by exploiting new hardware or operating system (OS) interfaces. Computer security attacks are also now relying on social engineering techniques to run malicious programs on victims' machines; instances of such abuse include phishing and watering hole attacks, both of which trick people into running malicious code or divulging confidential information. Thus, traditional computer security methods, such as OS confinement and program analysis, will not prevent new attacks that do not violate OS confinement or present illegal program behaviors.

Another challenge is that traditional security approaches have large trusted code bases (TCBs), which include hardware, OSs, and other software components that implement authentication and authorization logic across a distributed system. This is a vulnerable area because these components are complex and often contain vulnerabilities that undermine the overall system's integrity or confidentiality.

Evasive attacks on vulnerable systems – especially in instances where trusted components turn malicious – inspire the creation of defenses that can augment formally specified mechanisms against known threats. Specifically, this thesis advances the state of the art in *behavioral malware detection* – detecting previously unknown malware in the very early stages of infection within an enterprise network.

Here we assess three fundamental insights of modern-day attacks and then describe a cross-layer defense against such attacks. First, we make a low-level machine state visible to behavioral analysis, significantly minimizing the TCB and its associated vulnerabilities. Specifically, our behavioral detector utilizes an executable code's dynamic properties, with architectural and micro-architectural states as input. Second, we evaluate behavioral detectors against adaptive adversaries. For this purpose, we introduce a new metric to determine a detector's robustness against malware modifications, which serves as a step toward explainability of machine learning-based malware detectors. Finally, we exploit the fact that attacks spread through only a limited number of vectors and propose new techniques to analyze the resulting dynamic correlations created among machines. These insights show that behavioral detectors can efficiently protect both individual devices and end hosts within enterprise networks. We present three types of such behavioral

detectors.

Sherlock protects resource-constrained devices, such as mobile phones and Internet-of-things (IoT) devices, without modifying the software/hardware stack. Sherlock's supervised and unsupervised versions outperform prior work by 24.7% and 12.5% (area under the curve (AUC) metric), respectively, and detects stealthy malware that often evades static analysis tools.

The second behavioral detector, Shape-GD, protects devices within an enterprise network. It monitors devices on the network, aggregates data from weak local detectors, overlays that with network-level information, and then makes early, robust predictions regarding malicious activity. Shape-GD achieves its goals by exploiting latent attack semantics. Specifically, it analyzes communication patterns across multiple devices, partitioning them into *neighborhoods*. Devices within the same neighborhood are likely to be exposed to the same attack vector. Furthermore, we hypothesize that *the conditional distribution of false positives is different from that of true positives*; i.e., given a neighborhood of nodes, we can compute the aggregate distributional shape of alert feature vectors from the neighborhood itself and provide robust labels.

We evaluate Shape-GD by emulating a large community of Windows systems using the system call traces from a few thousand malicious and benign applications; we simulate both a phishing attack in a corporate email network as well as a watering hole attack through a popular website. In both scenarios, Shape-GD identifies malware early on ($\sim$100 infected nodes in a $\sim$100K-node system for watering hole attacks, and $\sim$10 of $\sim$1,000 for phishing attacks) and robustly (with $\sim$100%

global true-positive and $\sim$1% global false-positive rates).

The third behavioral detector, Centurion, detects malware across machines monitored by an anti-virus company. It is able to analyze behavior from 5 million Symantec client machines in real time and discovers malware by correlating file downloads across multiple machines. Compared with a recent local detector that analyzes metadata from file downloads, Centurion reduced the number of false positives from $\sim$1M to $\sim$110K and increased the true-positive rate by a factor of $\sim$2.5. In addition, on average, Centurion detects malware 345 days earlier than commercial anti-virus products.

Table of Contents

List of Figures

Chapter 1

Introduction

Used for everything from voting machines and elections to personal health and finances, computing is deeply and critically embedded in our society. However, due to malware's exponential growth and ability to quickly adapt to deployed defenses, best-known security techniques quickly become obsolete. To add a last line of defense against unknown 'zero-day' malware, we study behavioral malware detection – an attack-agnostic detection approach that either detects malware's deviation from a regular system's behavior (anomaly detection) or learns to classify the difference between benign and malicious programs' behaviors (malware classification).

1.1 Broader Impact

Computer systems today process billions of transactions per day in the financial industry; they drive electronic commerce; they store and process healthcare data; they manage key elements of the cyberphysical infrastructure (power plants, factories, etc.); participate in decision-making processes. Privacy and integrity are therefore paramount for much of the data with which computers work. Any software vulnerabilities can make such systems an easy target for attackers.

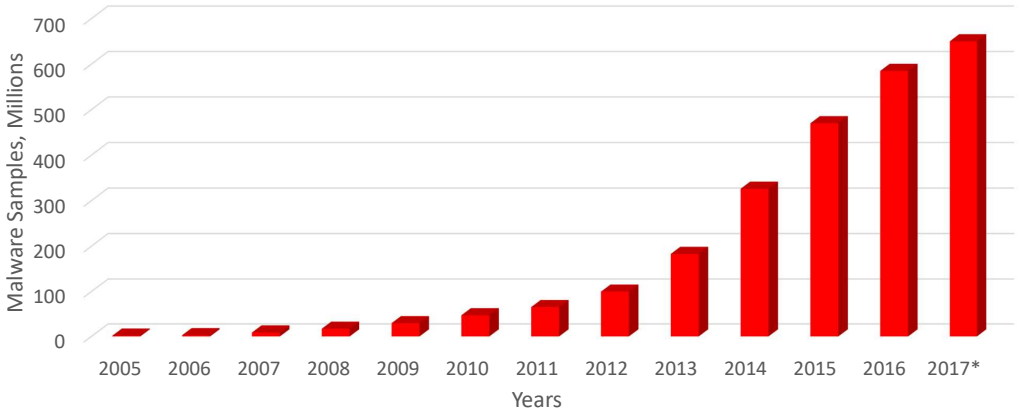

Figure 1.1: The cumulative number of malware samples registered by AV-TEST Institute. Malware has been growing very fast over the last decade.
*contains results obtained on 07/04/2017.

Recent malicious attacks [7, 22, 41, 45, 51] demonstrate how fragile the systems are and how few resources are needed to cause significant damage. In many cases, the damage can be beyond just financial loss. Take the example of the Democratic Party's breach [22] in 2016 or the multiple recent medical record breaches, such as the one at the Office of Personnel Management [21].

1.2 Cybersecurity Statistics

Despite enormous efforts by malware researchers and security practitioners, the malware population has grown rapidly. Malware adapts to the security solutions deployed in practice, and its developers find new ways of abusing emerging technologies, such as in mobile and IoT devices.

According to statistics released by AV-TEST [29], an independent German organization that periodically evaluates and rates antivirus and security suite soft-

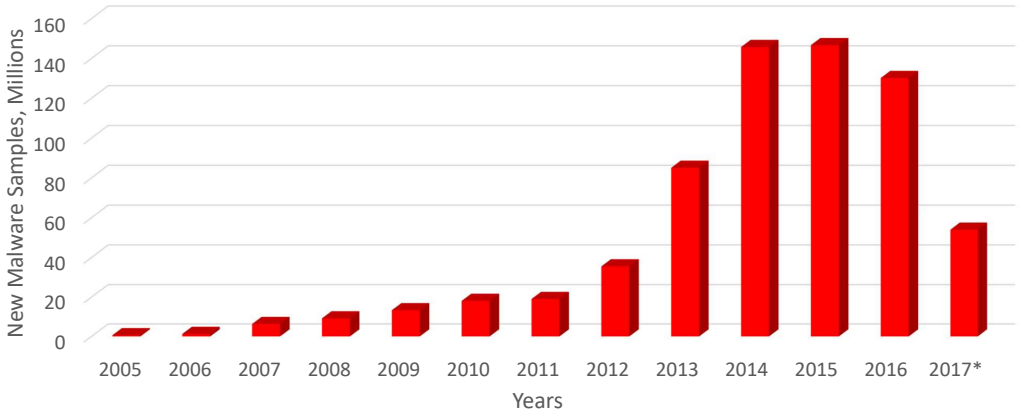

Figure 1.2: The number of new malware samples released every year.
*contains results obtained on 07/04/2017.

ware, the total malware population has grown from 1.7 million 'samples' (i.e., unique binaries) in 2005 to 650 million samples in July 2017 (Figure 1.1). Also we can notice that over the last four years, cybercriminals have significantly increased the rate of malware production. Rates have reached an average of 12 million new malware samples per month, in comparison to 52.5 thousand in 2005 (Figure 1.2).

Today, 85% of malware targets the Windows OS, as it is the most widespread and thus the most lucrative platform for attackers. However, this number is shrinking as more and more malware samples move into the mobile market. It is worth noting that mobile malware demonstrates a higher growth rate than its desktop counterpart. For example, the most prevalent type of mobile malware, Android malware, has grown from $\sim$360 thousand (January 2013) to $\sim$16.5 million (September 2016). This is not surprising because almost all applications that were available mainly on desktops only a few years ago (e.g., email, online banking) now conveniently function on mobile devices. The lack of adequate security solutions for mobile devices together with manufacturers' unwillingness to patch known Android

vulnerabilities make Android users easy prey for cybercriminals.

Attackers pursue multiple goals, from hijacking computer systems to gathering users' behavioral patterns to serve highly personalized ads. A large fraction of desktop malware still tries to hijack computers and join them to a botnet. Some malware hunts for online banking accounts, credit card data, and passwords. Information stealing is especially prevalent among mobile devices because the devices accompany the user everywhere and their applications can access a large number of sensors.

A recent trend is the rapid growth of potentially unwanted applications that collect information about users' surfing habits and other personal data and send users' profiles to advertising companies. Another trend is the growing number of ransomware, which is used to encrypt personal data on both desktop systems and mobile devices and extort money for releasing a decryption key.

Such infections are in fact enabled by commercial malware detectors that heavily rely on identifying malware signatures. Signature-based defenses are a reasonable way of protecting a system against trivial malware, but they do not work against polymorphic and metamorphic malware because they only capture the syntactic view of malware, i.e., the concrete way it is implemented. They can be generalized to cover similar malware variants, but self-modifying code can easily defeat such signatures.

To increase detectors' robustness to self-modifying malware, detectors must employ behavioral malware detection, which captures malware semantics rather

than syntactic properties. Later in this thesis, we describe how to design an end-host security solution that is based on the principles of behavioral malware detection.

1.3 Evolution of Attacks

The landscape of malicious attacks does not remain constant over time; rather, it evolves as new technologies emerge. At a high level, we can distinguish at least three broad categories of security attacks: memory corruption attacks, web attacks, and *modern attacks* – a new class of attacks that includes abuse of new interfaces and extensive use of social engineering. Attackers first started extensively exploiting memory errors in the 1990s (e.g., buffer overflows, double free, etc.), plenty of which were present in the then-dominant memory-unsafe languages. Attacks then moved to exploitating high-level web vulnerabilities, which opened an unprecedented way of directly executing high-level actions (e.g., exfiltrating sensitive information from a database, escalating user privileges in a web-based system). As developers started using memory-safe languages and security experts eliminated easily exploitable vulnerabilities, the attacker community added to its arsenal a powerful new technique: social engineering. *Social engineering* refers to psychologically manipulating people to perform actions or divulge confidential information. This type of attack mostly relies on human errors, whereas the other two solely exploit software vulnerabilities. Because they stem from human errors, social engineering attacks are hard to detect or prevent through purely technical means.

The most prevalent forms of malicious social engineering are phishing and

watering hole attacks. *Phishing* is a common method for obtaining private information by mimicking an email from a legitimate business such as a bank, credit card company, e-commerce retailer, etc. A phishing email usually contains either a link to a fraudulent web page that looks exactly like a legitimate one or a malicious attachment. A victim is usually asked to update some private information by following the malicious link, and thus attackers obtain that sensitive data. In the case of a malicious attachment, when an unsuspecting user opens it, malicious code is executed. It may collect private information in the background or even install a backdoor on a device.

The other popular social engineering attack technique, the *watering hole attack*, exploits users' trust in websites that they visit on a regular basis. Usually attackers start by gathering information about websites the user often visits from the secure system. Next, attackers search for vulnerabilities on those websites and inject code that can compromise the users' machines. The injected code supplies malware to the users upon connection and configures malware based on the users' environment to maximize its success rate. Typically, one or more members of the target group are compromised, and after that an attacker can easily move laterally within the secure system and compromise other machines [11].

In addition to social engineering techniques for attack, modern attacks extensively abuse new interfaces or use side channels, which are hard to preemptively discover and close. For example, recently released Spectre [37] and Meltdown [31] abuse speculative execution that leaves a secret-data-related micro-architectural state in the CPU after executing code that is not supposed to be executed. Both attacks

use a flush-reload [159] side channel to analyze the cache state modified by a speculatively executed code. Therefore, Spectre is able to leak victims' confidential information, and Meltdown is capable of breaking all security assumptions given by the address space isolation as well as paravirtualized environments and, thus, every security mechanism built upon this foundation. Another example, RowHammer [105, 135, 145, 152], exploits DRAM technology to flip bits in memory and is able to achieve privilege escalation.

1.4 Existing Approaches

In this section, we briefly summarize common approaches to mitigating software security issues. A detailed discussion of the related research is available in Chapter 5.

Security solutions can be classified into two broad classes based on their design characteristics: static and dynamic. The former analyze static properties of an artifact (e.g., a file, a network packet) that are potentially malicious without executing it, while the latter perform real-time analysis when a system runs with test inputs or when it is already deployed. *Static analysis* describes static code analysis that is extensively used to search for vulnerabilities in software. Such algorithms are usually designed to achieve soundness – meaning they never miss any bugs. However, soundness and completeness are not achievable simultaneously (because this is an undecidable problem). Usually, an incomplete static code analysis leaves behind many false positives that must be manually verified. Also, extensive use of dynamic features in modern languages such as dynamic types, dynamic method

resolution, and distributed and/or parallel algorithms significantly increase false-positive rates, thus limiting static analysis's real-world applicability.

Static analyzers may use machine-learning algorithms to detect malicious artifacts. They are not limited to code analysis; they may work with other types of artifacts such as network traces or API calls' or system calls' sequences. However, machine-learning systems' results usually suffer from poor explainability. Due to the use of multiple non-reversible transformations that lie at the heart of machine-learning algorithms, it is hard or sometimes even impossible to identify the root cause of an alert. Therefore, such detectors are not widely adopted in practice. Commercial solutions thus still heavily rely on manually hardcoded rules for bug detection to unambiguously map alerts to root causes.

The other class of methods, *dynamic methods*, analyzes program execution's side effects. This class can also be divided into two sub-classes. The first is based on runtime tracking of causality relations. Usually, it employs some form of dynamic information flow tracking to analyze how data is propagated through the code during execution. This can be relatively easy to achieve for managed languages such as Java, JavaScript, and Python by augmenting the just-in-time translator with a logic performing information flow tracking simultaneously with program execution. As for traditional compiled languages such as C/C++, the same effect can be achieved by inlining information-flow-tracking code into the original program. However, both methods usually introduce significant performance overhead, as they execute an information-flow-tracking engine in parallel with the original program.

The second subtype of dynamic analysis employs statistical methods. It

monitors how an application under test interacts with the rest of the system at one or more interfaces – for example, at system call, API, network, or memory allocation levels. A dynamic analyzer uses a model of a normal program behavior and compares real-time observations against predictions given by the model. It considers significant deviations from the model as a sign of malicious activity. In comparison to causality-based dynamic analysis, such detectors typically incur little overhead. However, they replace causality with a weaker property: correlation.

The other method for classifying existing approaches is to partition them into the following two classes: client-based and network-based. All approaches described above fall into the category of client-based detectors because they perform per-host analysis. An example of a network-based detector is a system that analyzes network communication patterns and the content of network packets. This example is quite promising because eventually almost all malware and other attacks spread over a network and thus are visible at the network level. However, encrypted traffic and the requirement to maintain low latency usually pose significant challenges for such detectors.

In practice, various combinations of such detectors can be used, and they are usually accompanied by ad hoc rules that match simple malicious behaviors observed in the real world.

Regardless of the method being used, it is often hard to distinguish between benign and malicious behaviors because malware can try to mimic benignware, which leads to high false-positive rates. In practice, to minimize false-positive rates, malware detectors are configured to be permissive, i.e., only highly suspi-

cious activity leads to an alert. Therefore, a significant portion of malware may remain undetected for a long period of time.

1.5 Why Machine Learning for Computer Security

Traditional approaches to computer security such as OS confinement, formal methods, and software verification often fail to protect systems against modern attacks for several reasons. First, such attacks may not be visible at the level of traditional malware detectors because new hardware or APIs can be exploited. Second, rather than directly compromising a system, attacks might trick the user into performing an action on an attacker's behalf. Therefore attacks do not violate any constraints that are enforced by traditional malware detectors. Third, software verification typically is not able to mitigate security issues because it is undecidable, it poorly scales up to large code bases, and real-world systems often lack precise specifications.

On the other hand, machine-learning-based malware detectors possess some appealing properties. First, they do not require any formal specifications. Second, machine-learning algorithms learn by example – they require data corresponding to normal and abnormal system behaviors, which is much easier to collect than to formally specify security properties and verify that a system complies with them. Third, such detectors can easily scale up to model practical systems of an arbitrary complexity and size.

However, they can only detect attacks or malware that exhibit statistical behavior at the given level of abstraction. In other words, if malware causes an

application to slightly deviate from a normal behavior, a machine-learning detector may ignore it if such a deviation is statistically insignificant. Therefore, a behavioral detector should operate on multiple layers of abstraction to prevent a large spectrum of attacks.

In summary, a machine-learning-based approach to computer security is one of the promising ways to develop low-overhead malware detectors that are capable of protecting large systems.

1.6 Overview of Our Solution

In this work, we address the key aspects of applying machine learning to computer security – explainability, a relatively high false-positive rate, and robustness to evasive malware – and present an end-to-end security solution that operates at multiple layers of abstraction on end-host devices and at the network level.

First, we develop a highly efficient behavioral malware detector that can be deployed even on resource-constrained devices such as mobile phones and IoT devices. The detector analyzes the executed code's dynamic properties and compares them with predictions made by state-of-the-art machine-learning models. If there is a deviation from the model, a detector raises an alert.

We introduce a new methodology for evaluating malware detectors – a white-box methodology together with an operating range concept. The white-box methodology allows us to understand malware internals rather than treating them as a black box. It includes a deep malware introspection and ensures correct malware execu-

tion in a laboratory environment. Also, it includes a method for developing evasive malware that is used to comprehensively evaluate a detector.

An operating range of a detector characterizes its detection capabilities with respect to a particular malware type. We can formally define it as the smallest malware payload X hidden in an application Y that a detector can detect with a false-positive rate of Z. In other words, for a chosen application Y and a fixed threshold Z, the operating range indicates the smallest malware payload that the detector can reliably detect. The operating range accompanies the white-box methodology because it is required to perform malware dissection in order to identify malware types. Also, it serves as a step toward explainability of machine-learning-based malware detectors.

Second, we develop a generic framework for aggregating predictions made by individual weak detectors (detectors having low precision) running on end-host devices and correlating them with network-level activity to drive the false-positive rate toward zero. Specifically, end-host detectors report data samples that lead to raising local alerts to a global detector. It reanalyzes multiple data samples simultaneously, allowing for recovery of precision at a global level. The main advantage of such an approach is its ability to reanalyze multiple local data samples overlayed with the network-level communication patterns. The network-level information is used to formulate a hypothesis about potential attacks, and the local data samples are used for hypothesis testing.

The global detector that we introduce in Chapters 3,4 relies on the difference of statistical shape of local detectors' false and true positives. Though a local

detector is not able to distinguish between individual false and true positives, the global detector can differentiate between them by aggregating multiple false and true positives and approximating the distributions that they come from. We empirically verified this assumption across multiple systems, both mobile and desktop, and across multiple OSs. Moreover, the assumption held for a large variety of local detector types.

In a practical setting, local detectors generate both false and true positives because some nodes on a network are compromised and others are not. The high number of false positives hides true positives and makes it hard for the global detector to distinguish between them using their statistical shape. To make malware more noticeable and thus facilitate early and robust malware detection, we introduce dynamic neighborhoods that aggregate nodes exposed to similar attack vectors. These can be used in conjunction with statistical shape. We use the neighborhood concept to increase malware concentration.

Malware spreading along one of the attack vectors compromises one or more neighborhoods. Thus, malware concentration (and the concentration of true positives) within such neighborhoods in a network is much higher than average. Consequently, the statistical shape of compromised neighborhoods is significantly different from the statistical shape of uncompromised neighborhoods.

We discuss both concepts – statistical shape and dynamic neighborhoods – in detail in Chapters 3,4, where we describe different types of global detectors.

1.7 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 outlines the design of end-host detectors, establishes general guidelines for the development and evaluation of such detectors, and shows how the principles work in practice. Chapter 3 focuses on the problem of aggregating multiple weak behavioral detectors to achieve a desirable false-positive level and presents detection results for watering hole and phishing attacks. Chapter 4 presents an end-to-end malware detector that performs efficient real-time malware detection among 5 million Symantec clients' systems and significantly outperforms a prior work in terms of malware-detection capabilities. Chapter 5 describes related work. Finally, Chapter 6 summarizes the contributions of this thesis and outlines future research directions.

Chapter 2

Quantifying and Improving the Efficiency of Hardware-based Mobile Malware Detectors[1]

2.1 Introduction

Mobile devices store personal, financial, and medical data and enable malicious programs to spread quickly through app-stores. Unsurprisingly, 2015 saw 900K new mobile 'malware' compared to 300K in 2014. Mobile malware infects applications through errors by users, developers, or platforms like Android [30, 47, 77]. Once infected, malware can run 'payloads' such as stealing private data from the victim device or making HTTP requests to attack a remote server while masquerading as the infected application. Hence, machine learning classifiers that differentiate operating system and network *behaviors* of benign programs from malware are an attractive line of defense against mobile malware [126].

Hardware-based malware detectors (HMDs) are a recent category of behavioral malware detectors [72, 103, 122, 142]. An HMD observes programs' instruction and micro-architectural traces and raises an alert when the current trace's

[1] Based on the research paper "Quantifying and improving the efficiency of hardware-based mobile malware detectors" [102] by M. Kazdagli, V. Reddi, and M. Tiwari published at 49th Annual IEEE/ACM International Symposium on Microarchitecture. M. Kazdagli was the lead researcher for the project.

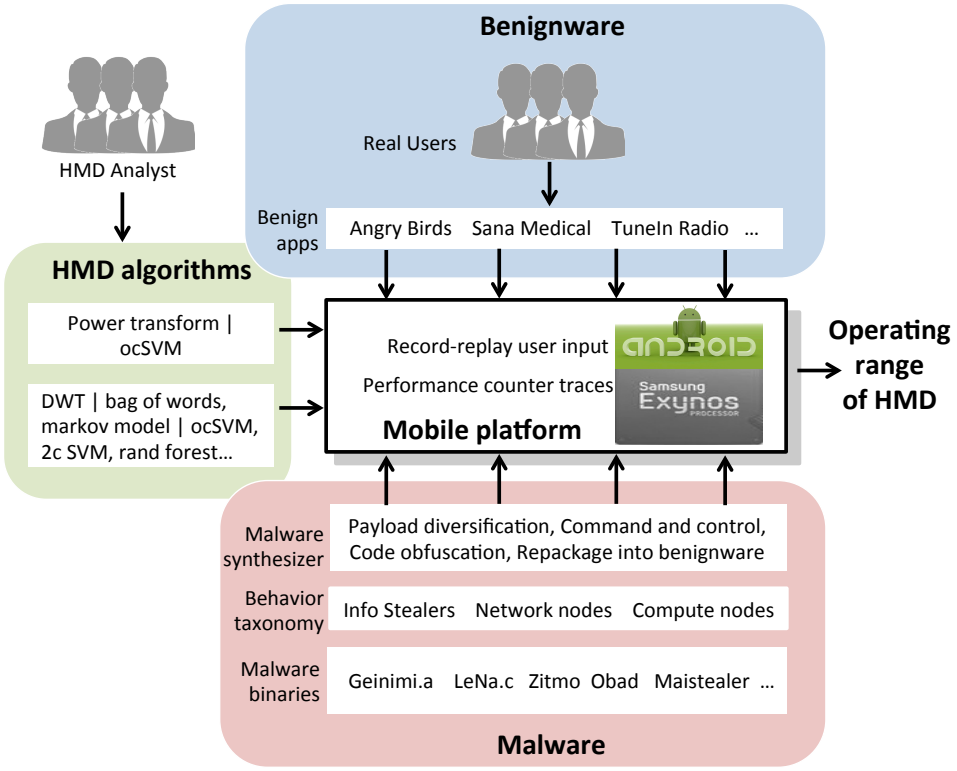

Figure 2.1: Overview of Sherlock.

statistics look either anomalous compared to benign traces (unsupervised HMDs) or similar to known malicious traces (supervised HMDs). HMDs are small, secure even from a compromised OS, and can observe instruction-level attacks (such as row hammer [105, 135, 145] and side-channel data leaks [121]) that leave no system call trace [125]. HMDs are thus a trustworthy first-level detector in a network-wide malware detection system [146, 165] and are being deployed in commercial mobile devices[2] as of early 2016.

[2]https://www.qualcomm.com/products/snapdragon/security/smart-protect

Architects designing HMD accelerators face two unique challenges. First, unlike benign programs like SPEC, malware *adapts* to proposed modeling algorithms and will evade detectors that only learn behaviors of existing malware [156]. For example, we found that changing only the number of execution threads or inter-action delays in a malware was sufficient to evade an HMD [72] trained on single-threaded binaries of the same malware. Second, to compare malware against benignware executions, HMDs have to run both programs with *real user inputs*. For example, a human user playing Angry Birds produces very different instruction traces compared to quiescent traces when no user is driving the app. Hence, the traditional 'black-box' approach for evaluating HMDs – without looking deeply into malware computation and without real user interaction – yields results that will not hold in practice.

In this paper, we present Sherlock—a 'white-box' methodology to evaluate HMDs for mobile malware. Sherlock is built on two principles: (1) malware will adapt to evade detection, and (2) malware hides behind benign programs, and only by running both malware and benignware with real user-inputs can we determine whether an HMD can tell them apart. These principles lead to a significant system-building effort and to new insights about HMDs for mobile malware.

The Sherlock platform in Figure 2.1 embodies both principles: (1) Sherlock synthesizes malware specifically to find the breaking point of an HMD under test. To do so, we introduce a taxonomy of mobile malware and present a synthesis tool that generates obfuscated malware with a configurable payload (i.e., tasks to run) that is a superset of the 229 malware we studied. (2) Sherlock tests HMDs when

benign and malware programs use the same, long-running user inputs. To do so, Sherlock correctly records and replays thousands of 5–10 minute long user sessions (such as playing Angry Birds or running medical diagnostics) on real hardware. An HMD analyst can then use Sherlock's third component – HMD algorithms – to design and evaluate new ways of extracting features from program traces in order to train machine learning algorithms.

Sherlock's design principles yield a new metric for quantifying HMDs' performance. An *operating range* of an HMD algorithm is a metric that tells an analyst the root cause behind a malware alert as well as when the HMD fails. An operating range is expressed as *the smallest malware payload X hidden in application Y that an HMD algorithm A can detect with a false positive rate of Z.* For example, an analyst can determine how efficiently a compromised browser (Y) can steal SMSs or photos (X) when a random-forest HMD (A) is deployed at a pre-set false positive rate of 5% (Z).

The operating range of an HMD is independent of the training and testing set of malware – instead, it is defined in terms of atomic actions in malware payloads (X) such as stealing one photo or an SMS, sending an HTTP request, etc. An analyst can thus use the operating range to quantify HMD performance based only on (relatively invariant) high-level malware behaviors. Further, operating range describes false positive rate Z by comparing malware to the exact benign app Y that malware hides in—comparing a malware run to an arbitrary benign app or system utility yields an unrealistically good false positive rate.

Case Studies using Sherlock. We demonstrate Sherlock's utility by designing bet-

ter HMDs than prior work, and by showing (for the first time) that evading static program analysis makes malware more visible to HMDs.

Our first case study shows that taking concrete mobile malware actions into account yields a better unsupervised HMD than directly applying desktop HMDs designed to detect short-lived exploits. Specifically, atomic software-level actions on mobile devices such as stealing a 4MB photo or one SMS takes a long time at the hardware level (2.86s and 0.12s respectively on a Samsung Exynos 5250 device). We design a new HMD that uses longer-duration (100ms) feature vectors, extracts low-frequency signals, and is 24.7% more effective using the area under the ROC curve (AUC) metric than prior work [142].

Our second case study uses Sherlock's malware synthesizer to design supervised HMDs with 97.5% AUC – 12.5% better than prior work. Specifically, we train on a malware set that covers diverse, orthogonal behaviors compared to prior work that trains HMDs on an ad-hoc subset of behaviors. Further, the supervised HMD's operating range covers even small data (1 photo, 25 contacts, 200 SMSs, etc) being stolen with close to 100% accuracy at a 5% false positive rate. However, malware payloads that clog remote servers by sending them HTTP requests are virtually undetectable at the hardware level—Sherlock provides such semantic insights into why HMDs succeed and fail.

Our final case study in using Sherlock's malware synthesizer yields a surprising result—*obfuscation techniques that evade detection by static analysis tools make HMDs more effective.* Specifically, malware developers use string encryption and Java reflection to create high-fanout nodes in data- and control-flow graphs and

thus foil static analysis tools. However, these obfuscation techniques in turn create instruction sequences and indirect jumps that make malware stand out from benignware. Hence, light-weight HMDs can complement static analysis tools [42] used by Google and other app stores to drive malware down into more inefficient settings.

In summary, we make the following contributions:

1. Malware Synthesis. We deconstruct 229 malware binaries from 2013–2015 to create a malware synthesis tool. An analyst can use the synthesis tool to determine an HMD's operating range.

2. Record-and-replay Platform. Sherlock records and replays 1–2 hours each of real human input for 9 benign applications and over 69 hours across 594 malware binaries. Without correct replay at these time-scales, malware payloads will not execute to completion.

4. Three case studies with new insights. We improve HMDs' performance by 24.7% and 12.5% respectively for unsupervised and supervised HMDs and show that HMDs detect stealthy malware that evades static analysis tools.

Sherlock Detailed information on how to reproduce the experiments can be found at https://github.com/Sherlock-2016.

2.2 Motivation

Before we dive into the details of Sherlock in Sections 2.3 and 2.4, we begin with the unique advantages of HMDs over OS-level detectors and challenges in evaluating HMDs.

2.2.1 HMDs in a Network of Weak Detectors

HMDs (as well as OS-level detectors) are deployed in a collaborative intrusion detection system (CIDS) that has two components. On the server side, a platform provider (e.g., Google) executes benign and/or malware applications using test and real user inputs, measures instruction statistics (using performance counters for example), and creates a database of computational models. On client devices, a light-weight *local detector* samples performance counters to create run-time traces from applications, compares each run-time trace to database entries on the device, and forwards suspicious traces to a *global detector* on the server.

Importantly, HMDs do *not* need to have 0% false positives and 100% true positives—they only need to serve as an effective filter for a global detector that can then use program analysis [56, 75] or network-based algorithms [71, 153] to build a robust global detector. We refer readers to Vasilomanolakis et al. [146] for a survey on collaborative malware detectors.

2.2.2 Hardware vs OS-level detectors

HMDs are more trustworthy, light-weight, and hard to hide from compared to detectors that use system call [62, 126], middleware [52], or network based behavioral analysis [98],

HMDs' are trustworthy since they can be isolated from most of the OS (and Android middleware) and run inside a hardware-based enclave [53, 94] or directly in hardware [122] – secure against even user errors and kernel rootkits [30]. HMDs can be battery efficient with feature extraction and detection logic implemented

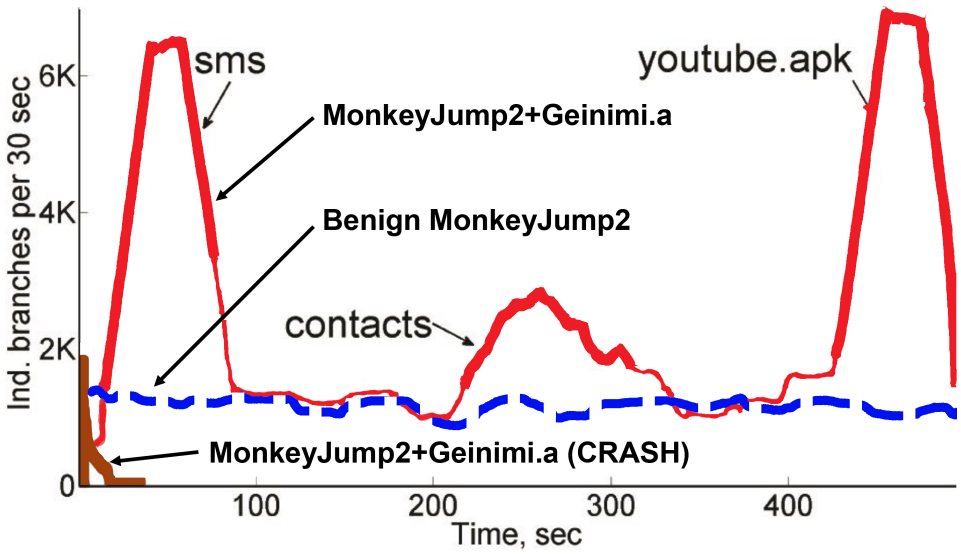

Figure 2.2: Executing malware payloads. The off-the-shelf Geinimi.a malware crashes immediately. Once fixed, Geinimi.a executes malicious payloads such as stealing SMSs or contacts or downloading files.

using accelerators [122]. Finally, HMDs can detect malware that leaves *no* system call trace – such as rowhammer [105, 135], A2 [158], and side-channel attacks [125] on desktops and, as we show in this paper, evasive mobile malware that hides behind benign applications and requests no additional sensitive permissions.

Interestingly, on mobile platforms, HMDs have comparable detection rates to OS-level detectors (although we leave details of this comparative experiment out of this paper). We find that OS-level detectors that model system calls also reach detection rates of almost 90% at false positive rates close to 10%. This is close to our HMDs' performance (in Section 5)—as a result, HMDs can not just be more trustworthy than OS-level detectors but be competitive in detection performance as well.

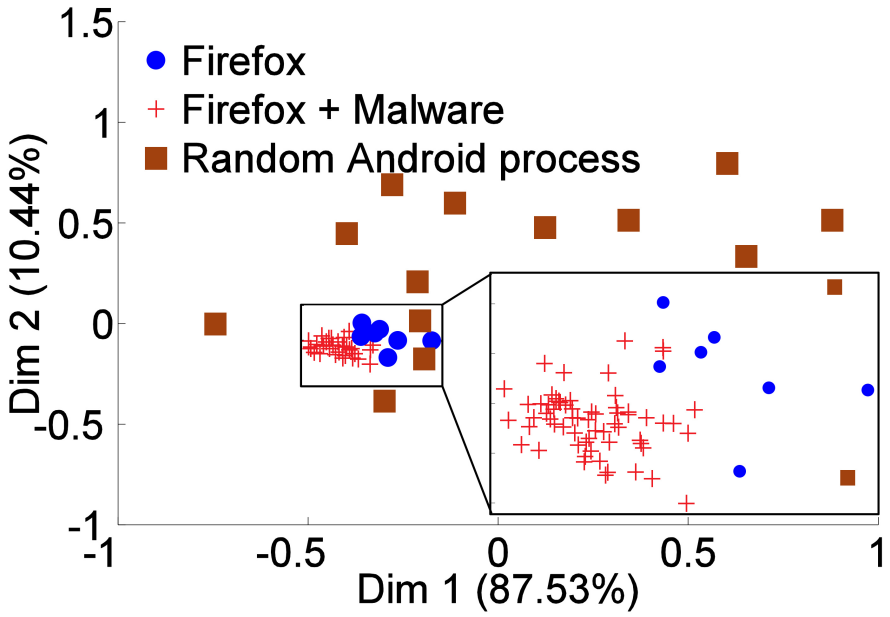

Figure 2.3: Differential analysis of malware v. benignware. The plot shows principal components of benign Firefox, Firefox with malware, and arbitrary Android apps. Malicious Firefox's traces are closer to Firefox than to random apps.

HMDs for desktops do not directly port over to mobile platforms. Ozsoy et. al's [122] hardware-accelerated classifiers detect ∼90% of off-the-shelf desktop malware with 6% false positive rate. Tang et. al's [142] anomaly detector achieves 99% detection accuracy for less than 1% false positives on a set of PDF and Java malware. Such desktop HMDs, however, do not work well for mobile platforms – we quantify Tang et. al's HMD against mobile malware in our first case study in Section 5 and build a 24.7% better HMD using Sherlock.

2.2.3 Challenges in Evaluating HMDs

The closest related work to ours – on HMDs for mobile malware – is by Demme et al. [72], where the authors present a supervised learning HMD that compares off-the-shelf Android malware to arbitrary benign apps, yielding a 90:10 true positive to false positive ratio. However, this methodology of using off-the-shelf malware and comparing it to arbitrary benign apps is fallacious, as we discuss next.

Adaptive malware. One challenge in evaluating detectors is that malware developers can *adapt* their apps in response to proposed defenses. For example, we find that simply splitting a payload into multiple software threads dramatically changes the malware's performance-counter signature and training a supervised HMD on the single-threaded execution yields a very low probability of labeling the multi-threaded version as malware. Adding delays, changing payload intensity, or choosing an alternative victim application also throws off a supervised HMD trained only on traces from existing malware.

Prior work analyzes malware samples categorized by family names like `CruseWin` and `AngryBirds-LeNa.C`—this does not tell an analyst why a malware binary was (not) detected. Instead, we propose to determine *why* a particular malware sample was (un)detectable, to anticipate *how* it can adapt, and then to create a malware benchmark suite to *identify the operating range* of the detector.

Correct execution. The second challenge is that mobile malware samples available online [33, 166], and used in prior work, seldom execute 'correctly'. Malware often require older, vulnerable versions of the mobile platform, they may target specific

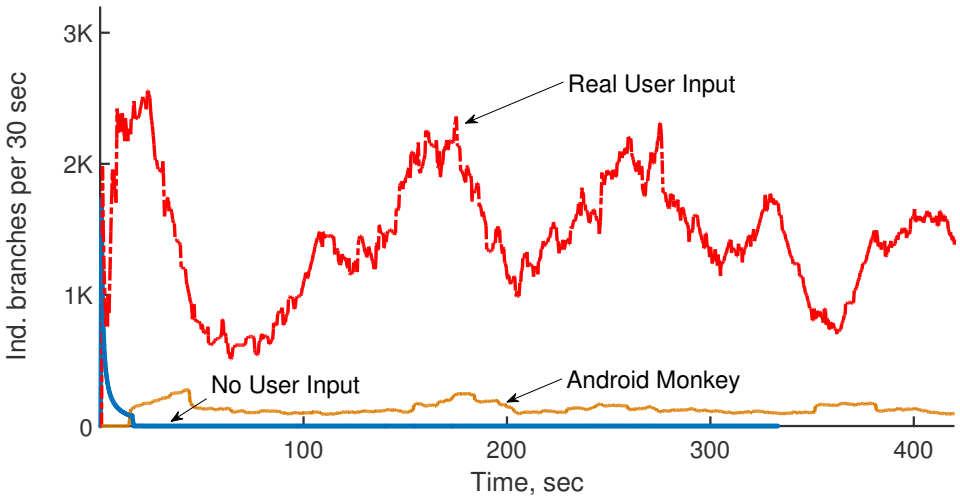

Figure 2.4: Real user inputs create hardware level activity, while providing no input or using Android's input-generation tool (Monkey) creates a very small signal.

geographical areas, include code to detect being executed inside an emulator, wait for a (by now, dead) command-and-control server to issue commands over the internet or through SMSs, or in many cases, trigger malicious actions only in response to specific user actions [3, 15].

Figure 2.2 shows that an off-the-shelf malware (Geinimi.a) simply crashes on our Android board and thus looks distinct from a benign MonkeyJump game's trace. Prior work will mis-classify this as a true positive. 20% of malware executions in Demme et al's [72] experiments lasted less than one second and 56% less than 10 seconds – in comparison, stealing a single photo takes almost 3 seconds. Instead, we ensure that malware executes 'correctly' – steals SMSs and contacts, and downloads an app – and aim to identify these payloads.

Appropriate Benignware and Real User Inputs. The third challenge is to ensure

appropriate differential analysis between benign and malware executions. Prior work [72] trains detectors on malware executions but tests against *arbitrary* benign applications. However, a benign app infected with malware looks more similar to the underlying benign app than an arbitrary benign app. Figure 2.3 plots the execution traces of Firefox, Firefox with malware, and randomly chosen Android processes along the first two principal components that retain $\sim 99\%$ of the signal. We see that the infected Firefox traces are much closer to those of benign Firefox than to any other Android process like `netd`. Hence, false positive rate of an HMD for Firefox should be tested using a benign Firefox – testing the HMD against arbitrary processes [72] will yield wrong results that favor the HMD.

Further, Figure 2.4 shows that driving Android applications using real user-input (red curve) has a major impact on the execution signals compared to giving no input (blue curve) or using the Android 'Monkey' app (light brown curve) to generate random inputs. Behaviors with 'no inputs' or 'Android Monkey' (blue and brown curves) can be easily captured by a behavioral detector, and, as in the previous case, this leads to overestimation of its actual detection performance. Hence, we propose to test HMDs using malicious binaries against appropriate benign apps while both apps are being driven using real user-inputs.

Quantitative Comparison to Prior Evaluation Methods. We have shown in this section that prior 'black-box' methods yield traces that do *not* represent either malware or benignware executions. The prior method has logical flaws – as a result, 20% of malware traces in [72] are shorter than 1 second, and 56% are $<$10s – and we deliberately eschew further quantitative comparisons with Sherlock. Instead, our

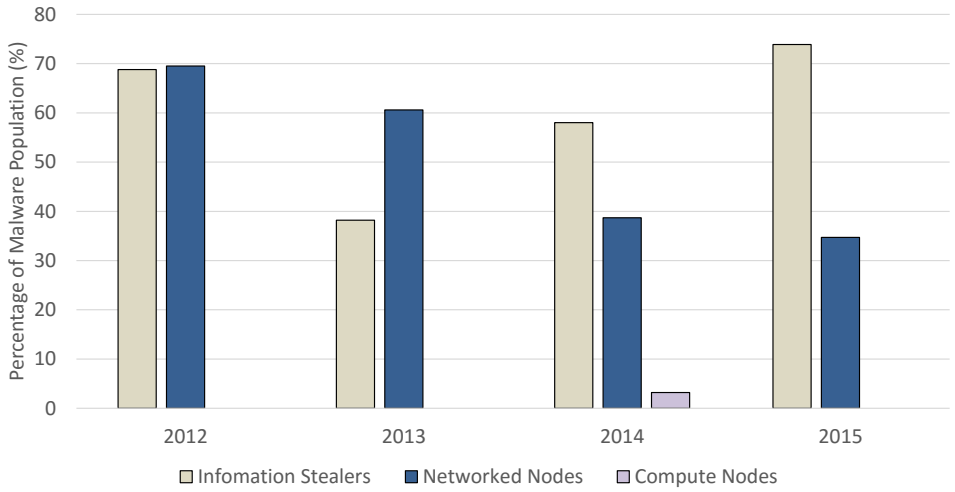

Figure 2.5: Malware behaviors observed in a 126-family 229-sample Android malware set from Contagio minidump. Most malware steals data or carries out network fraud. However, samples that use phones as compute nodes, e.g., to crack passwords or mine bitcoins, have been reported in 2014.

evaluation focuses on case studies using Sherlock to yield new insights into building effective HMDs.

2.3 Synthesizing Mobile Malware

The first major component of Sherlock generates a diverse population of malicious apps. To do so, we first introduce a taxonomy of high-level malware behaviors, and then use it to create a set of representative malware whose hardware signals have been explicitly diversified.

Figures 2.5 and 2.6 show our manual classification of malware into high level behaviors. We studied 53 malware families from 2012, 19 from 2013, 31 from 2014 and 23 from 2015 – a total of 229 malware samples in 126 families –

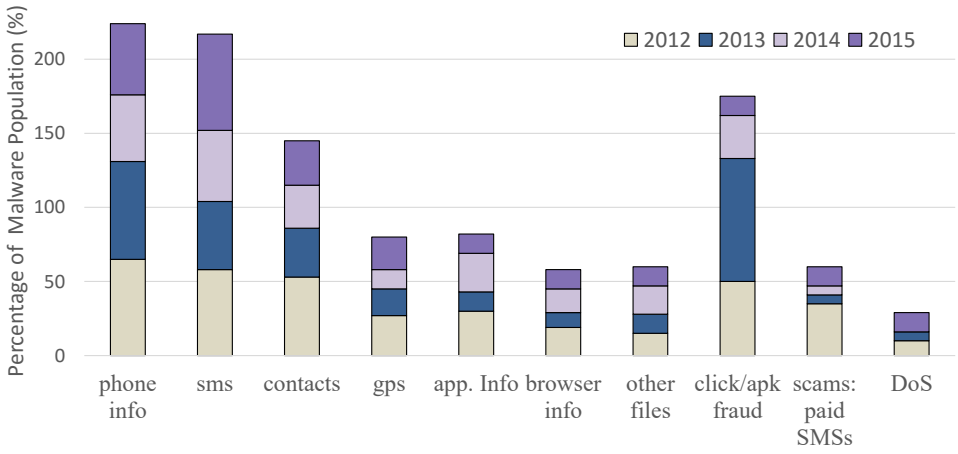

Figure 2.6: Examples of malware behaviors and their contribution to the malware dataset.

downloaded from public malware repositories [27, 28, 33]. Our classification's goal is to identify orthogonal atomic actions and to determine concrete values for these actions (e.g., amount and rate of data stolen).

To classify malware, we disassembled the binaries (APKs on Android) and executed them on both an Android development board and the Android emulator to monitor: permissions requested by the application, middleware-level events (such as the launch of Intents and Services), system calls, network traffic, and descriptions of malware samples from the malware repositories. We describe our findings below.

2.3.1 Unique Aspects of Mobile Malware

Our key insight is that instead of trying to detect conventional root *exploits* [13, 16, 44], we propose to detect malicious *payloads*. Here, payloads refer to code that achieves the malware developers' goals, such as sending premium SMSs,

stealing device IDs or SMSs, etc. We observed root exploits in only 10 of 143 samples in 2012 and 3 of 32 samples in 2013 – we now take a closer look at the attack vectors mobile malware rely on.

Mobile malware can successfully execute payloads due to vulnerable third-party libraries. In one instance that affected hundreds of millions of users, a "vuln-aggressive" ad-library had a deliberate flaw that led to downloaded files being executed as code [47]. Webviews, that enable Android apps to include HTML/-javascript components, are another major source of vulnerabilities [66] that allows payloads to be dropped to a device. Apps with this vuln-aggressive library or Webviews are otherwise benign and can be downloaded from app stores as developer signed binaries, only to be compromised when in use.

In other cases, errors by an app's benign developers themselves can lead to malicious payloads being executed. Misconfigured databases even in popular apps like Evernote [12] and AppLocker [5] (a secure data storage app) were vulnerable to malicious apps on the device simply reading out data from sensitive databases. In such cases, the malicious app could be an otherwise harmless wallpaper app that constructs an 'Intent' (a message) to AppLocker's database at run-time and exfiltrates data if successful.

User errors are another cause for malware payloads executing successfully at run-time. Malicious apps read data from an online server, use it to construct a user prompt at run-time, and thus request sensitive permissions such as access to SMSs or microphone. Users often accept such requests [76] and once authorized, apps can siphon off *all* SMSs or conduct persistent surveillance attacks [4].

Worst of all, even the platform (Android) code can have severe vulnerabilities that doesn't require a conventional exploit. For example, the Master Key vulnerability [30] simply involved an error in how Android resolves a hash collision due to resource-names in a binary at install time v. execution time. By packing the binary with a malicious payload such that the install time check passes but the execution time loader picks the other malicious payload, attackers could distribute their payloads through signed apps in official app-stores.

Finding: analyze payloads instead of exploits. We conclude that while there are many routes to getting a payload to execute as part of a benign app, executing the payload is mandatory for malware to win. Hence mobile HMDs shoud aim to *distinguish malicious payloads from benign app executions*. The challenge of detecting payloads is that payloads can look very similar to benign app's functionality. For example, if a previously harmless AngryBirds game starts to comb through a database, can we distinguish whether it is reading a user's gaming history (harmless) or a user's SMS database (attack) using only hardware signals.

2.3.2 Behavioral Taxonomy of Mobile Malware

At a high level we assigned every malicious payload to one or more of three behaviors: *information stealers, networked nodes*, and *compute nodes* (Figure 2.6).

Information stealers look for sensitive data and upload it to the server. User-specific sensitive data includes contacts, SMSs, emails, photos, videos, and application specific data such as browser history and usernames, among others. Device-specific sensitive data includes identifiers – IMEI, IMSI, ISDN – and hardware and

network information. The volume of data ranges from photos and videos at the high end (stolen either from the SD card or recorded via a surveillance app) to SMSs and device IDs on the low end.

The second category of malicious apps requires compromised devices to act as nodes in a network (e.g., a botnet). Networked nodes can send SMSs to premium numbers and block the owner of the phone from receiving a payment confirmation. Malware can also download files such as other applications in order to raise the ranking of a particular malicious app. Click fraud apps click on a specific web links to optimize search engine results for a target.

Given the advances in mobile processors, we anticipated a new category of malware that would use mobile devices as compute nodes; for instance, mobile counterparts of desktop malware that runs password crackers or bitcoin miners on compromised machines. This was confirmed by recent malware that mines cryptocurrencies [32]. We use a password cracker as a compute-oriented malware payload. The cracker's task is to recover sensitive passwords by making a guess, compute the guess' cryptographic hash, and compare each hash against a secret database of hashed passwords.

Finding: Software-level actions are surprisingly long in hardware. Figure 2.7 shows the specifics of each malware behavior we currently include in Sherlock. Interestingly, *atomic* malware payload actions take significant amount of time at the hardware level for several payloads – e.g., stealing even one SMS or a Contact requires 0.12s to 0.36s on average. These constants inform the design of our performance counter sampling durations and machine learning models in Section 2.4. The

Synthetic Malware	Parameters (number of items)	Malware-Specific Delay (ms)	# of RPKG Mal. Apks	Length per Action (sec)	Inst. Count (Million)
Steal files (4.2MB each)	1, 15, 35, 50	0, 1K, 5K	12	2.86	50.97
Steal contacts	25, 70, 150, 250	0, 10, 25	12	0.36	67.80
Steal SMSs	200, 400, 700, 1.7K	0, 15, 40	12	0.12	25.90
Steal IDs, GPS	data size fixed	0, 200	2	4*	39.65
Click fraud (pages)	20, 80, 150, 300	0, 1K, 3K	12	0.40	44.40
DDos (slow loris)	500 connections	1, 40, 80, 200	4	425	49.70
SHA1 pass. cracker	10K, 0.5M, 1.5M, 2.5M	0, 20, 40	12	2.8E-5	1.9E-2

Figure 2.7: Malware payloads: 4 info stealers, 2 networked nodes, and 1 compute node. These settings represent a small but computationally diverse subset of malware behaviors. Interestingly, small software actions have large hardware footprints.

last two columns in Figure 2.7 show the average length of an atomic action in the malware payload (not counting delays such as being scheduled out by the operating system), and the instruction count per action (e.g. stealing 1 photo/contact/SMS, clicking on 1 webpage in click fraud, opening 500 connections and keeping them alive in a DDoS attack, generating 1 string and computing its hash using SHA1).

2.3.3 Constructing Malware Binaries

We now describe the steps required to create a realistic malware binary. Malware activation can be chosen from being triggered at boot-time, when the repackaged app starts, as a response to user activity, or based on commands sent over TCP by a remote command and control (C&C) server. In all cases, malware communicates back to the C&C server to transfer stolen data or compute results. Sherlock's configuration parameters also specify network-level intensity of malware

App name	Description	# of Installs	User Actions	User Time (min)	CPU Time (min)	Inst. Count (Billion)
Amazon	internet store	10M – 50M	searched for sporting goods; looked through 25 pages; clicked on 50 items	81.15	32.40	1,914.97
Angry Birds	game	1M – 5M	played 9 rounds and completed 7 levels	76.97	63.76	1,047.73
CNN	news app	5M – 10M	browsed several categories of news and a few articles of each type	58.04	11.60	254.85
Firefox	browser	50M – 100M	browsed 20 webpages starting from google.finance	93.96	45.51	1,464.52
Google Maps	map service	500M – 1B	browsed maps of a few cities and opened street views	56.09	35.38	768.31
Google Translate	translator	500M – 1B	translated 30 words, searched history, tried handwriting recognition	59.72	12.12	203.61
Sana MIT Medical	medical app	U/A	completed 5-6 questionnaires	111.41	11.37	145.94
TuneIn Radio	internet radio	50M – 100M	switched amongst 6 channels and listened to radio	78.10	26.17	407.99
Zombie WorldWar	game	1M – 5M	played 5 rounds and completed 4 levels	91.62	88.40	2,261.99

Figure 2.8: Real user inputs on benign apps, with per app traces up to $\sim$2 hours and $\sim$2 trillion instructions. We choose complex apps and include a mix of compute (games), user-driven (browsers, medical app), and network-centric (radio) apps.

payload in terms of data packet sizes and interpacket delays, and device-level intensity in terms of execution progress (in terms of malware-specific atomic functions completed). We chose concrete parameters for malicious payload based on an empirical study of mobile malware as well as information about benign mobile devices [101].

The generated malware has a top-level dispatcher service that serves as an entry point to the malicious program; it parses the supplied configuration file, launches the remaining services at random times, and configures them. Malicious services can run simultaneously or sequentially depending on the configuration pa-

rameter. In some cases, the service that executes a particular malicious activity can serve as an additional dispatcher. For example, the service executing click fraud spawns a few Java threads to avoid blocking on network accesses. Every spawned thread is provided with a list of URLs that it must access. Besides Android services, we register a listener to intercept sensitive incoming SMS messages, forward them to C&C server, and remove them from the phone if needed. This listener simulates bank Trojans that remove confirmation or two-factor authentication messages sent by a bank to a customer.

Most professional apps are obfuscated using Proguard [17] to deter plagiarism. Proguard shrinks and optimizes binaries, and additionally obfuscates them by renaming classes, fields, and methods with obscure names. We applied Proguard to the malware payloads (even when we did not use reflection and encryption) to make the payloads look like real applications.

After a malware payload is created, it must be repackaged into a baseline app. Repackaging malware into a baseline app involves disassembling the app (using `apktool`), and adding information about new components and their interfaces in the application's Manifest file. We then insert code into the Main activity to start the top-level malware dispatcher service (whose activation trigger is configurable), and add malicious code and data files into the apk. We then reassemble the decompiled app using `apktool`. If code insertion has been done correctly, apktool produces a new Android app, which must be signed by `jarsigner` before deployment on a real device.

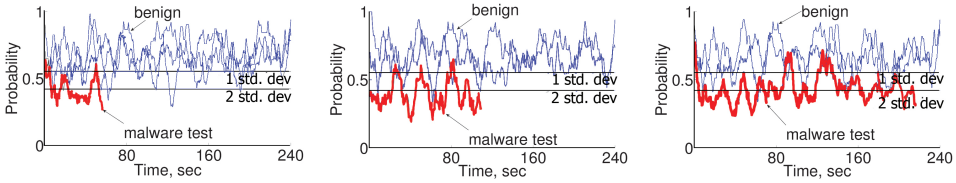

Figure 2.9: HMD results for Angry Birds with click fraud operating at three (increasing) intensities. Since HMD is trained on benign AngryBirds, a low dark-line shows that the HMD detects malware as a low probability state.

2.4 Real User-driven Execution

Armed with a computationally diverse malware suite, we now select a similarly diverse suite of benign apps, drive them with long, real, user inputs, and extract hardware signals from them. Figure 2.8 shows the apps that we run along with their inputs – using these, we find that since the apps' traces are so diverse, we need to build HMDs customized for each app.

2.4.1 Benign Apps

Our main goal is to choose applications that represent popular usage, and that require permissions to access resources like SD card and internet connectivity. This ensures that the applications are interesting targets for malware. Further, we ensure that the apps cover a mix of compute (games), user driven (medical app, news), and network (radio) behaviors, diversifying the high-level use cases for apps in the benignware suite. Our chosen app set includes native (C/C++/assembly), Android (Dalvik instructions), and web-based functionality, varying the execution environment of our benign app pool. We confirm that this high-level diversity does indeed translate into diverse hardware-level signals.

2.4.2 User Inputs

For each benign application, we created a workload that represents common users' behavior according to statistics available online. For example, when exercising Firefox, we visited popular websites listed on alexa.com. Automating this is simple. For Angry Birds, we recorded a user playing the game for multiple rounds and successfully completing several levels. For the medical diagnostics app (Sana), we record users completing several questionnaires, where each questionnaire requires stateful interactions spread over several screens. Such deep exploration of real apps is far beyond the capability of not only the default UI testing tool in Android (Monkey [43]), but also state of the art in input generation research [114]. Without such deep exploration of benign apps, the apps' hardware traces will reflect only a dormant app and cause the malware signals to stand out at test time but not in a deployed system.

For each benign app, we collect 6 user-level sessions (each 5–11 min long) and use a heavily modified Android Reran [85] to record and replay 4 of these sessions with random delays added between recorded actions (while ensuring correct execution of the app). These 10 user-level traces per app generate 56–111 minutes of performance counter traces across all apps.

Each benign app is then repackaged with 66 different payloads to create 9 $\times$ 66 malware samples. To collect performance counter traces, we replay one of the app's user-level traces and extract 5–11 minutes long performance counter traces for *each* malware sample.

Figure 2.8 shows some interesting trends in benign traces. While Sana commits 145 Billion instructions in 111 minutes, Zombie WorldWar commits 2,261 Billion instructions in 91 minutes – clearly, Sana is much more user-bound while Zombie WorldWar is compute-heavy. CNN and Angry Birds are similar to Zombie WorldWar, where TuneIn Radio lies between Sana and Zombie WorldWar in instructions committed.

Finding: HMDs have to be application-specific. Interestingly, as we show in our evaluation (Section 2.5), the compute intensity of CNN and Zombie WorldWar results in them having the worst detection rates among all the apps in our suite. On the other hand, even though TuneIn Radio is more intense than Sana, TuneIn Radio exposes malware better. We find that this is because the Radio has more regular behavior while Sana executes in short, sharp bursts. Sherlock's realistic replay infrastructure and user-input traces are key to producing these insights into HMDs' performance.

2.4.3 Extracting Hardware Signals

We now describe our measurement setup for precise reproducibility. The measurement setup requires careful setup and correctness checks since it is difficult to replay real user inputs to the end once delays and malware payloads are added.

Devices. Our experimental setup consists of an Android development board connected to a desktop machine via USB, which in turn stores data on a server for data processing and construction of ML models. The desktop machine uses a wireless router to capture internet traffic generated by the development board. The traffic

collected from the router is analyzed to ensure that benignware and malware execute correctly.

We use a Samsung Exynos 5250 equipped with a touch screen, and a TI OMAP 5430 development board, and we reboot the boards between each experiment. We ran all experiments on the Exynos 5250 because some common apps like NYTimes and CNN crashed on OMAP 5430 for lack of a WiFi module, but repeated Angry Birds experiments on the OMAP 5430 to ensure that our results are not an artifact of a specific device.

Performance counter tracing. We used the ARM DS-5 v5.15 framework and the Streamline profiler as a non-intrusive method for observing performance counters. DS-5 Streamline reads data every millisecond and on every context switch, so it can ascribe performance events to individual threads. However, in DS-5 Streamline extracting per process data can only be done using its GUI – we automate this process using the JitBit [23] UI automation tool.

Choice of performance counters. We used hardware performance counters to record five architectural signals: memory loads\stores, immediate and indirect control flow instruction counts, integer computations, and the total number of executed instructions; and one micro-architectural signal: the total number of mispredicted branches. We collected counter information on a per process basis as matching programmer-visible threads to Linux-level threads requires instrumenting the Android middleware (i.e., is non-trivial), and because per-process counters yielded reasonable detection rates. We leave exploring the optimal set of performance counters for future work.

Overhead of counter sampling. We found that sampling counters with 1 ms time resolution incurs less than 0.3% slowdown on the CF-Bench mobile benchmark suite. Prior work also reports low overheads: Demme et al report 5% overhead at 25k cycles per sample, while Tang et al report 1.5% at 512k retired instructions per sample. Beyond sampling, the detection logic itself is fairly simple – while we describe our HMDs in the next section, Ozsoy et. al have shown that a neural net HMD costs less than 5% in area and delay to an AO486 CPU (with overheads expected to be smaller for larger CPUs).

Ensuring correct execution. We ensured that the malicious payload was executed correctly on the board for each trace. Specifically, synthetic malware communicated with a Hercules 3-2-6 TCP server running on the desktop computer, which recorded a log of all communication. The synthetic malware itself printed to a console on the desktop computer (via `adb`) as well as to DS-5 Streamline when running each malicious payload.

For experiments with off-the-shelf malware, we developed an HTTP server to support custom (reverse-engineered) duplex protocols for C&C communication. If we allowed malware to communicate to its original server, which was not under our control, we captured network traffic going through the router. We checked the validity of performance counters readings obtained via DS-5 Streamline with specially crafted C programs, which we compiled and ran natively on the boards.

2.4.4 Constructing and Evaluating HMDs

Using benign and malware traces collected as described above, an HMD analyst can then train and test a range of HMD algorithms. For example, Figure 2.9 shows one of the HMD algorithms we present in a case study in Section 2.5.1. The HMD is an anomaly detector and the figure plots the likelihood that the current trace is going through a known phase—a low probability thus indicates potential malware (the dark line) while higher probabilities indicate benignware (light gray lines). Increasing the payload's intensity lowers the probability even further. By tuning the probability at which a time interval is flagged as malicious (or by training a classifier to learn this), an analyst can trade-off false positives and true positives.

Importantly, we evaluate true positives and the detection threshold using only the time windows that contain malware payload execution. We do *not* use time windows where our repackaging code and dispatcher service executes, since we would like the HMD to be evaluated solely using payloads and not exploits. We do not use time windows *before* or *after* the payload is complete, because if an HMD raises an alert when the payload is *not* executing, the alert may in reality be a false positive that will get recorded as a true positive. Prior evaluation methods do not separate out malware payload intervals and may have this error. On the other hand, to measure false positives, we use benign traces only and hence use the entire trace durations for each experiment. Finally, we use 10-fold cross validation on an appropriate subset of our data to evaluate HMDs.

2.5 Case Studies using Sherlock

We show how malware analysts can use Sherlock through three case studies. **(1)** We use malware payload sizes in Section 2.3 to tune the machine learning features (100ms v. sub-ms in prior work) for an unsupervised HMD. Our HMD outperforms prior work designed to detect short-lived exploits by 24.7% on the area under curve (AUC) metric (Section 2.5.1). **(2)** Sherlock's taxonomy of malware in Section 2.3 can be used to train a supervised learning based HMD eficiently. This 'balanced' HMD outperforms alternative HMDs – that are trained on subsets of malware behaviors – by 12.5% AUC when tested on new variants of the behaviors. **(3)** Surprisingly, we show that our unsupervised HMD can detect malware that uses obfuscation to evade the best known static analyses. Hence, HMDs and static analyses are complementary and can drive malware payloads towards inefficient implementations.

2.5.1 Improving Unsupervised HMDs

We begin by quantifying why prior work designed to detect exploits may not yield the best HMDs to detect long-lived payloads.

Exploit-based ML features do not expose payloads (Figure 2.10). Tang et al. [142] present an HMD specifically designed to detect the multi-stage exploits that characterize Windows malware. The HMD samples performance counters every 512k cycles, and uses a power transform on performance counter data to separate benign and malicious time intervals. Then, a one-class SVM (ocSVM) is trained on short-lived features – i.e., on each sample as a non-temporal model and using 4

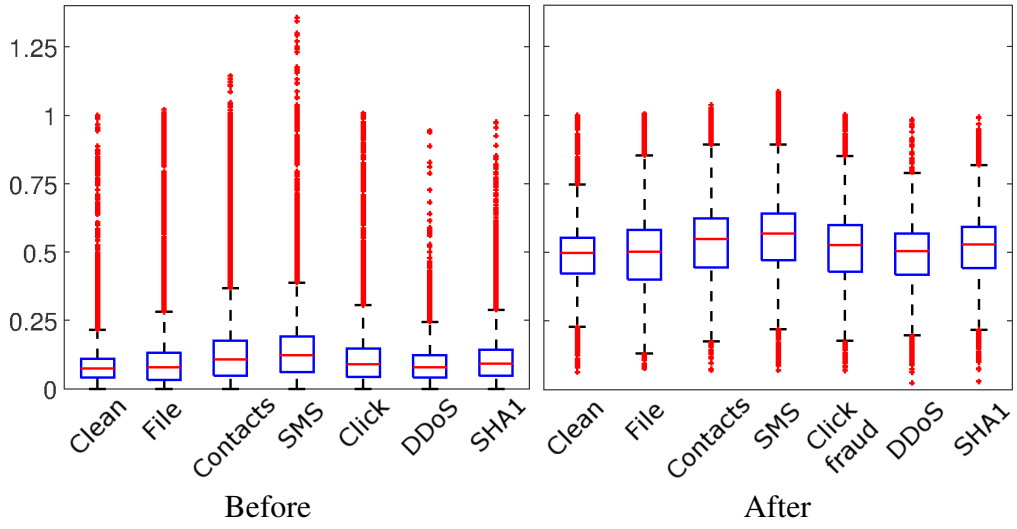

Figure 2.10: Distribution of load/store events in Angry Birds before and after power transform. Power transform does not make malware *payloads on Android* more discernible from benign behavior, whereas Tang et al. [142] show that it separates *exploits* from benign apps in Windows.

consecutive samples to train a temporal model – to label anomalous time intervals as malicious.

We find that power transform does *not* have the same effect on mobile malware payloads—payloads look very similar to benignware traces even after a power transform. For example, Figure 2.10 shows the distribution of load-store instruction count per time interval for benign Angry Birds (labeled 'Clean'), compared to time intervals in Angry Birds infected with different malware payloads (e.g., file stealer, click fraud, DDoS, etc)—before and after a power transform. The distributions are shown as a box-and-whiskers plot, where the box edges are 25^{th} and 75^{th} percentiles, the central mark is the median, the whiskers extend to the most extreme data points not considered outliers, while the outliers are plotted individually in

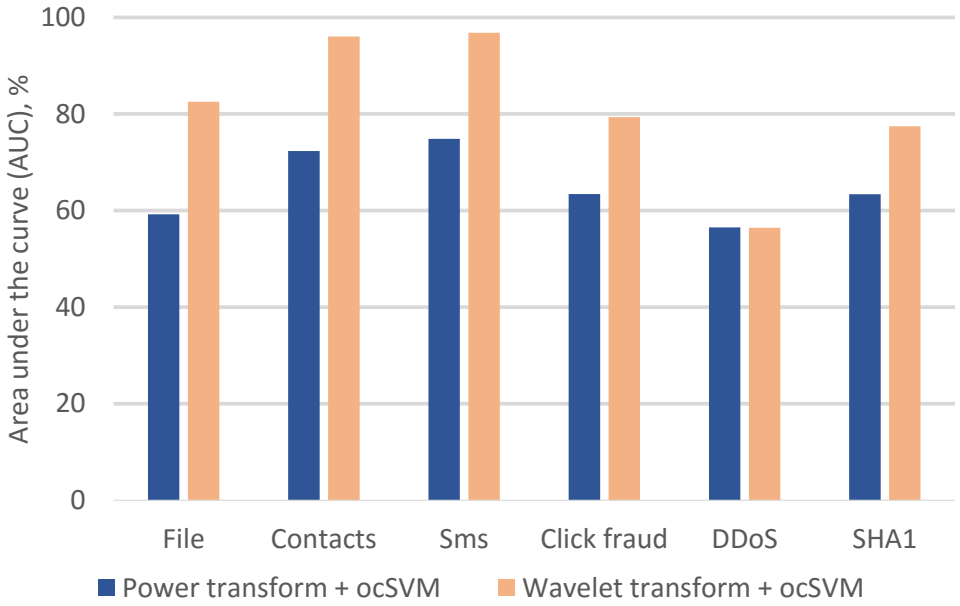

Figure 2.11: Comparison of power transform + ocSVM (prior work) and Discrete Wavelet Transform + ocSVM (this work). Our detector has 24.7% better area under curve metric (AUC) than prior work.

red. Data in both plots have been normalized to the range of benign Angry Birds' values. We use the standard Box-Cox power transformation to turn performance counter traces into an approximately normal distribution. Since the distributions of malware and benignware in Figure 2.10 overlap significantly, training an ocSVM on this dataset will yield a poor HMD as we show next.

Payload-centric ML features. We designed a new HMD whose features reflect our findings about mobile malware payload sizes in Figure 2.7. Specifically, we attempt to capture program effects at the scale of 100ms intervals, i.e., closer to the time required for atomic actions like stealing information or networking activity.

We then extract features from each 100ms long time interval using Discrete

Wavelet Transform (DWT) and use the wavelet coefficients as a feature vector for the time interval. The wavelet transform can provide both accurate frequency information at low frequencies and time information at high frequencies, which are important for modeling the execution behavior of the applications. We use a three-level DWT with an order 3 Daubechies wavelet function (db3) to decompose a time interval. We also used the Haar wavelet function with similar detection results.

Finally, we use multiple feature vectors to construct two models: (a) a bag-of-words algorithm followed by a ocSVM, and (b) a probabilistic Markov model. Both these models are simple to train and compute at run-time, and hence serve as good local detectors (and a good baseline for more complex models such as neural nets that are harder to train).

2.5.1.1 Bag-of-words Anomaly Detector

The bag-of-words model treats 100ms time intervals as words and a Time-to-Detection (TTD) window as a document. We experimented with a range of words and TTDs, finding a codebook of 1000 words and TTD = 1.5 seconds to yield good results. The bag of words algorithm maps each TTD window into a 1000-entry histogram, and trains a one-class SVM on benign histograms. We parameterize the one-class SVM so that it has $\sim$20% percent false positives.

Comparison with power transform — ocSVM HMD. Figure 2.11 compares our bag-of-words based ocSVM with one that uses a power transform using the area under ROC curve (AUC) metric. Note that AUC is a relative metric to compare classifiers, whereas the operating range measures an HMDs' robustness to atomic-action-

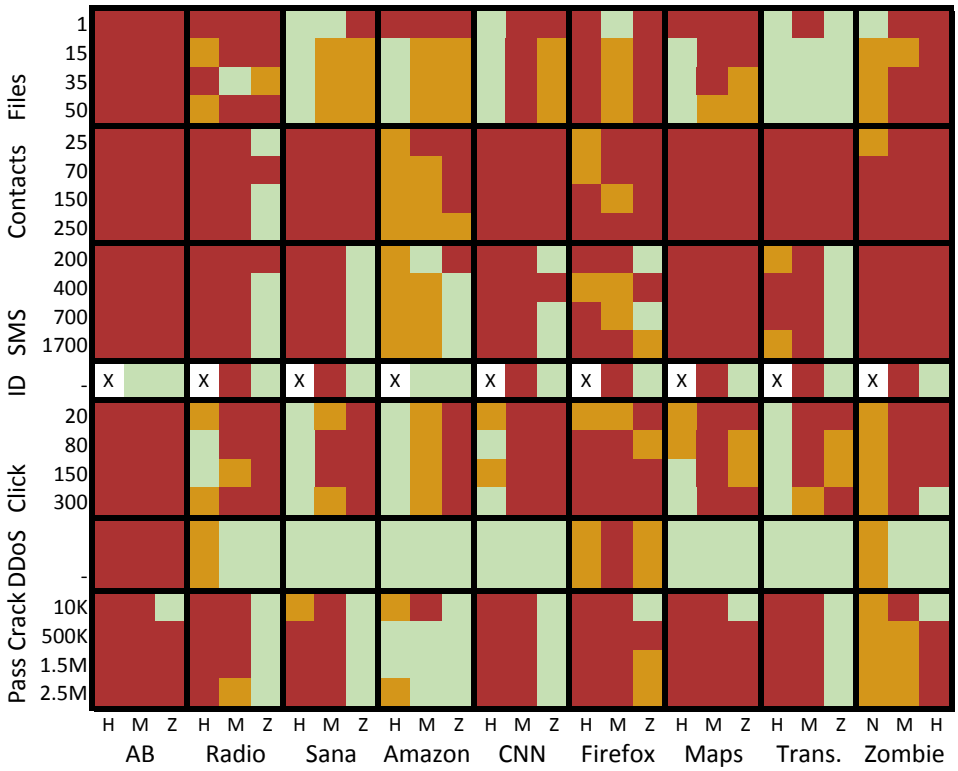

Figure 2.12: The operating range of Bag-of-words HMD. In each rectangle, the size of malicious payload grows from the top to the bottom, and the amount of delay decreases from left to right (H=High, M=Medium, Z=Zero delay). If color goes from light to dark within a rectangle, then the detection threshold (i.e., the lower end of the operating range) lies inside the rectangle.

sized mutations in malware. The bag-of-words model outperforms prior work for each category of malware behavior and by an average of 24.7% higher AUC across all malware.

Operating range of DWT — bag-of-words — ocSVM. Figure 2.12 shows the operating range for the bag-of-words model. Each cell in the matrix corresponds to a malware payload action (y-axis) and benign app (x-axis) pair. The malware pay-

loads are grouped by category and within each category, increase in size from top to bottom and in delay from right to left. These experiments use parameters from Figure 2.8. The intensity of the color – from light green to dark red – corresponds to the detection rate, which is computed as the number of raised alarms versus the total number of alarms that could be raised.

Figure 2.12 shows that the bag-of-words model achieves, at ∼20% false positive rate: 1) surprisingly high true positive rate for dynamic, compute intensive apps such as Angry Birds (99.9%), CNN (84%), Zombie WorldWar (93%), and Google Translate (92.4%); and 2) ∼80% true positive rate for both Amazon and Sana.

Bag-of-Words model HMD space overheads. Bag-of-Words models require 639KB – 1,229KB space with an average of 840KB and less than 2% of the average size of Android apps.[3]

2.5.1.2 Markov-model based Anomaly Detector

We present an alternative HMD to show that HMD models should be chosen specific to each application, and that there is an opportunity to apply ensemble methods to boost detection rates.

Our first-order Markov model based HMD assumes that the normal execution of an application (approximately) goes through a (limited) number of states (program phases), and the current state depends only on the previous state. The

[3]https://crowdsourcedtesting.com/resources/mobile-app-averages

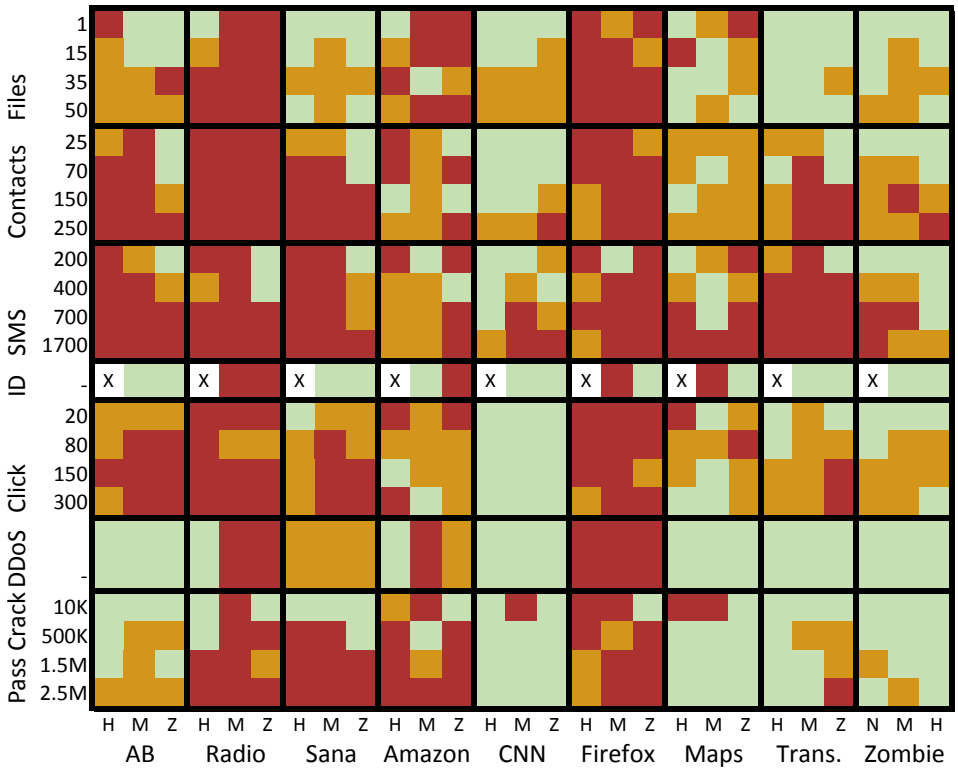

Figure 2.13: The operating range of Markov model HMD. Interestingly, the Markov model performs worse than the simpler bag-of-words model for compute intensive and dynamic apps (e.g., Angry Birds, CNN, and Zombie WorldWar).

goal is to detect malware if its performance counter trace creates a sequence of rare state transitions (as shown in Figure 2.9).

The HMD uses DWT to extract features as in the bag-of-words model, but maps them to a smaller number of words (i.e., states in the Markov model) using k-means clustering. We use the Bayesian Information Criterion (BIC) score [127] to find that 10 to 20 states is a good number across all benign apps. Using observed state transitions derived from training data, we empirically estimate the transition matrix and initial probability distribution (through Maximum Likelihood Estima-

tion). For detection, the Markov model HMD tracks the joint probability of a sequence of states over time and if malware computations create anomalous hardware signals (i.e. this probability is below a threshold for 5 states in our model), the HMD raises an alert.

Operating range of DWT — Markov model HMD. Figure 2.13 shows the results-matrix for the Markov model based detector. All the results are shown for a false positive rate of 20-25%.

Increasing the size of each payload action makes malware more detectable – this can be seen as the colors being more intense towards the bottom part of most rectangles. Increasing the delay between two malicious actions does not have a similarly predictable effect – SMS stealers in Angry Birds is a rare pair where detection rate increases with delay. This is interesting since intuitively, adding delays between payload actions should decrease the chances of being detected. However, these experiments indicate that for most malware-benign pairs, detection depends on how each payload action interferes with benign computation rather than delays between the payload actions.

The most important take-away from Figure 2.13 is that for most malware-benignware pairs, the detectability changes from light green to dark red as we go from top to bottom in the rectangle – this shows that our malware parameters in Figure 2.13 are close to the detection threshold, i.e. the lower end of the HMD's operating range for the current false positive rate. There are a few exceptions as well, such as click fraud, DDoS, and password crackers hiding in CNN; and DDoS in Angry Birds, Maps, Translate, and Zombie World Wards. For these cases, the

payload intensity has to be increased further to find their detection threshold.

Markov model HMD space and time overheads. Markov models representing the behavior of the benign apps vary from 1.2KB to 6.7 KB, with an average size of 3.2KB – they are thus cheap to store on devices and transfer over cellular networks. Its time to detection ranges between 1.2 seconds to 4.4 seconds and about 2.5 seconds on average. This means that the system can detect suspicious activities at the very beginning, considering that exfiltrating even one photo takes 2.86 sec on average.

2.5.1.3 HMDs Should Be App-specific

Interestingly, the Markov model works significantly better than bag-of-words for TuneIn Radio – with a 10% FP: 90% TP rate compared to 38%FP: 90% TP rate respectively – but performs significantly worse on apps like Angry Birds. In summary, a deployed HMD will benefit from choosing the models that work best for each application, but due to their different TP:FP operating points, will also benefit from using boosting algorithms in machine learning [134].

2.5.2 Improving Supervised HMDs

Sherlock can significantly improve performance of supervised learning based HMDs; specifically, by training the HMDs on a 'balanced' training data set that contains malware with each high-level behavior identified in the Section 2.3.2. Note that supervised learning techniques can be trained to recognize specific malware families [72] (i.e. a multi-class model) or to coalesce all malicious feature vectors

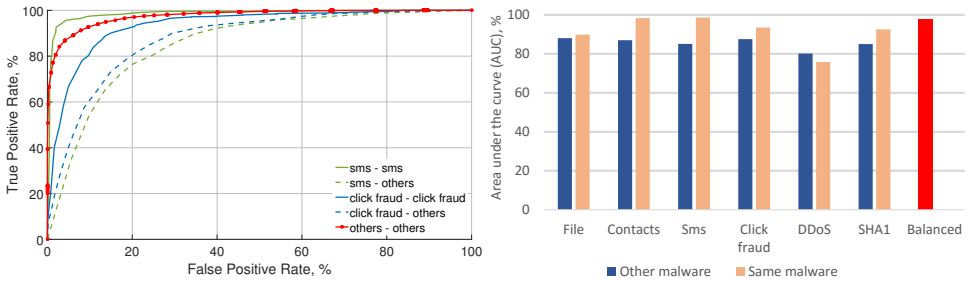

Figure 2.14: Training supervised learning HMD on a balanced set of malware behaviors yields best results.

(FVs) into a single label (i.e., a 2-class model)—we evaluate both categories in Figure 2.14.

To quantify Sherlock's 'balancing' effect on the resulting performance of a classifier, we conduct the following experiment. We partition the entire malicious set of FVs into a training set and two testing sets such that a training set contains malicious FVs of a particular type (e.g. SMS stealer, file stealer, DDoS attack and etc). The remaining FVs are placed in two non-overlapping testing sets. The first testing set includes the same type of FVs as the training set, while the second one comprises of FVs not in the training set. Finally, we add to each training/testing set benign FVs whose number is equal to the number of malicious FVs in the corresponding set.

Thus, for every partition we conduct two experiments: train a classifier on a training set and test it on the two testing sets. We present the results for Random Forest classifier using ROC curves (left) and AUC metric (right) in Figure 2.14 to compare relative performance of a classifier under different training and testing data sets. Each ROC curve is labeled as 'training malware type' – 'testing malware type'

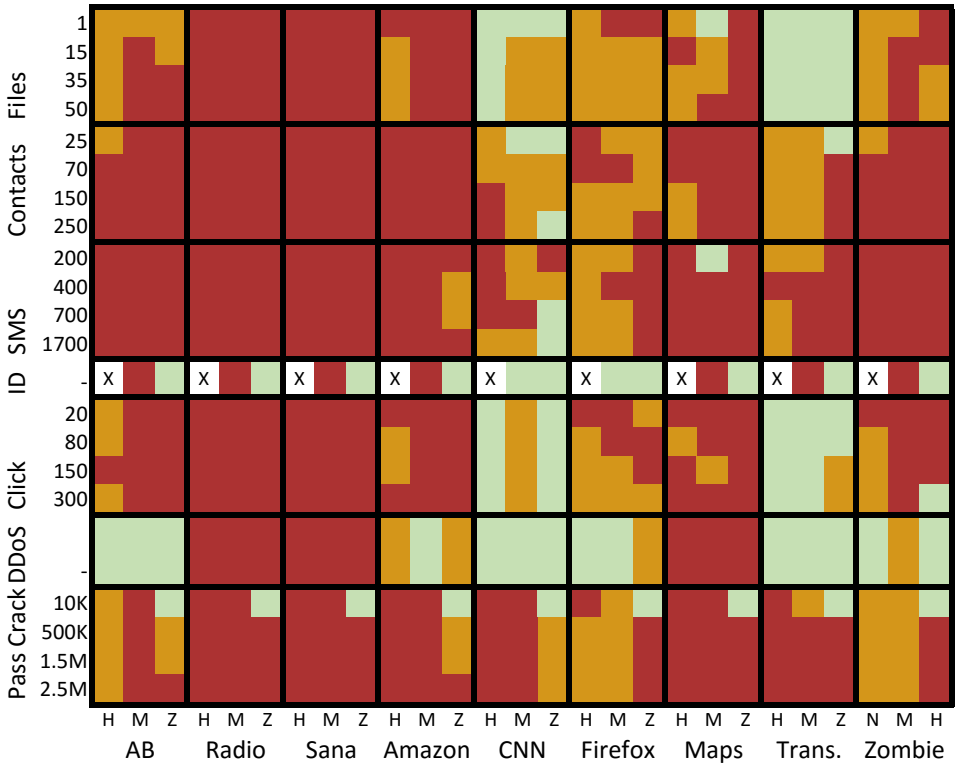

Figure 2.15: Operating range of 2-class Random Forest HMD: more effective than anomaly detectors when trained on a balanced dataset of all malware behaviors.

('others' means all malware types that are not included in the training set). And the red ROC curve shows the result of training and testing on a 'balanced' malware set.

We show relevant ROC curves in Figure 2.14 along with the AUC metric for all ROC curves on the right. The light brown bars correspond to AUC of the experiments where we train and test a classifier on the same malware type, i.e. we test it on the first testing set. The blue bars demonstrate classifier's 'cross' performance, i.e. we test it on the rest malware types (on the second testing set). All results are computed using 10-fold cross validation.

We experimented with several supervised learning algorithms – e.g., decision tree, 2-class SVM, k-Nearest Neighbor, Boosted decision trees, and Random Forest (RF)– and present the results for RF classifier because it demonstrated the best performance on our data set.

The common trend that we observed across all nine apps and all malware types is that the RF classifier has significantly better performance when testing on the same malware types (solid lines are higher than the dashed ones). The only exception is when the RF HMD is trained on DDoS malware, it surprisingly achieves better performance on other malware behaviors than on the in-class malware behaviors.

This can be explained by high stealthiness of our implementation of a DDoS attack [36] – each mobile device only opens HTTP connections to a target server and keeps them alive with minimal further requests. Thus, in real apps that also make network connections, DDoS *should* be virtually undetectable using HMDs. From machine learning perspective, DDoS and benign apps are likely closer to each other, while other malware like info-stealers and compute nodes are farther away in feature space, therefore we observe an opposite trend in the case of DDoS experiment in Figure 2.14.

Further, we trained a classifier on a balanced set of malicious data that included all malware behaviors in Sherlock. The solid line with dots (in the ROC plot) and the column on the far right (in the AUC bar graph) in Figure 2.14 show that showing some variants of each behavior enables the RF to achieve a higher detection rate (on even new variants) than both prior work as well as one-class SVMs.

The RF HMD can, for example, detect close to 85% of the malware with only 5% false positives compared to our anomaly detectors' similar true positives for $\sim$20% false positive rates. Finally, the RF HMD trained on a balanced data set yields 97.5% AUC whereas RF HMDs trained on per-behavior inputs yield AUCs of 91% and 85% when tested against the same or new malware behaviors respectively (averaged across all behaviors).

Operating range of Random Forest HMD. Figure 2.15 shows the detection results matrix for the RF HMD across the entire malware payload (Y-axis) and benignware (X-axis) categories for a fixed false positive rate of 5%. The key results are that RF detects most payloads except for detecting click fraud and DDoS attacks in CNN, Firefox, and Google Translate. It is likely that DDoS attacks – which involve a sequence of infrequent HTTP requests – look very similar to benign apps and are not well suited to be detected using HMDs. Indeed, all three HMDs – bag-of-words, Markov model, and RF – do a poor job of detecting DDoS attacks in most apps. On the other hand, RF consistently detects information stealers and compute malware (password cracker) across most apps. For apps with regular behavior (Radio) or sparse user-driven behavior (Sana), RF can detect all but the smallest of malware payloads.

In summary, Sherlock helps an analyst develop a robust HMD—first by dissecting existing malware to identify orthogonal behaviors, and then by training the HMD on a representative set of malicious behaviors. In the end, using the operating range, Sherlock informs the analyst of the type of behaviors the HMD is well/poorly suited at detecting.

```
1  Code snippet extracted from Obad.apk
2  Method: com.android.system.admin.
3  loOcccoC.loOcccoC(final boolean b)
4
5  dynamically construct class name
6  String class_name = oCIlCll(594, 24, -27);
7  return a class object
8  Class<?> c = Class.forName(class_name);
9  dynamically construct the name of a method
10 String method_name = oCIlCll(250, 33, -51);
11 return an object associated with the method
12 Method m = c.getMethod(method_name,
13           new Class<T>[] { Long.TYPE });
14 m.invoke(value, array);
```

Figure 2.16: Code shows Java reflection and string encryption in `Obad` malware that foils static analysis tools.

2.5.3 Composition with Static Analyses

Reflection is a powerful method for writing malware that evades static program analysis tools used in App Stores today [9]. Interestingly, we show that malware that uses reflection to obfuscate its static program paths in turn worsens its dynamic hardware signals, and improves HMDs' detection rates.

Java methods invoked via reflection are resolved at runtime, making it hard for static code analysis to understand the program's semantics. At the same time, reflection alone is not sufficient – all strings in the code must also be encrypted, otherwise the invoked method or a set of possible methods might be resolved statically.

To illustrate an actual malicious use of Java reflection and encryption, we show a code snippet (Figure 2.17) from `Obad` malware [3]. The code decrypts class and method names (lines 6 and 10) by calling the method `oCllCll()`. As a result, static analyses [56, 113] either do not model reflection or conservatively over-approximate the set of instantiated classes for `method_name` (line 10) and target methods for the `invoke` function (line 14). Due to control-flow edges that may never be traversed, static data-flow analysis becomes overly conservative, and static analyses end up with high false positive rates (or more commonly, with malware that goes undetected).

We augmented our synthetic malware with reflection and encryption similar to `Obad`'s implementation. Static analysis of our malware does not reveal any API methods that might raise alarms—we tested this using the Virustotal online service which ran 38 antiviruses on our binary without raising any warnings.

Figure 2.17 shows results of using the Markov model HMD on the 66 synthetic malware samples from Figure 2.7 augmented with reflection and encryption, and embedded into each of AngryBirds, Sana, and TuneInRadio. We see that in Angry Birds and Sana the detection rate of the malware that uses both reflection and encryption is significantly higher because reflection and encryption are computationally intensive and disturb the trace of the benign parent app (i.e., more than the same malware without reflection and encryption). We do not see the same trend for TuneInRadio because its detection rate was already quite high, so the additional impact of reflection on TuneIn Radio stays within the noise margin. We conclude that HMDs complement current static analyses and can potentially reduce the pressure

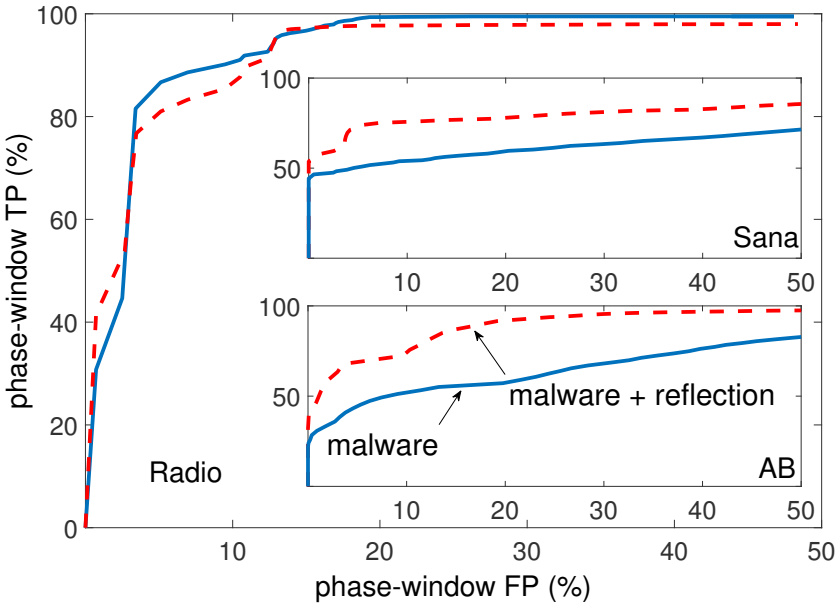

Figure 2.17: (Markov model) Effect of obfuscation and encryption on detection rate: interestingly, malware becomes more distinct compared to baseline benign app.

on computationally intensive dynamic analyses with a larger trusted code base [75].

2.6 Conclusions

HMDs are being studied by processor manufacturers like Qualcomm and Intel. As computer architects explore new hardware signals and accelerators to improve security in general and malware detectors in particular—our work lays a solid methodological foundation for future research into HMDs for mobile platforms. In particular, our approach of identifying *why* a detector succeeds and fails, instead of black-box experiments with malware binaries, is crucial. Indeed, prior

work has pointed out the pitfalls of using machine learning in a black-box manner for network-based intrusion detection systems [140]. Our future work will include applying Sherlock's white-box methodology to software detectors and efficiently composing them with HMDs.

Chapter 3

Early and Robust Malware Detection in Enterprise Networks

3.1 Introduction

Behavioral detectors are a crucial line of defense against malware. By extracting features out of network packets [89, 124, 140, 163], system calls [62, 83, 116], instruction set [68, 91], and hardware [72, 104, 143] level actions, behavioral detectors train machine learning algorithms to classify program binaries and executions as either malicious or benign. In practice, behavioral detectors are deployed extensively as per-machine local detectors whose alerts are analyzed by global detectors [1, 2, 19, 20, 34, 79].

However, behavioral detectors are *weak* – i.e., have high false positives and negatives. This is because a large class of malware includes benign-looking behaviors, such as encrypting users' data, use of obfuscated code, or making web/HTTP requests. Further, machine learning-based detectors have been shown to be susceptible to evasion attacks [123, 141, 156] that either increase false negatives or force detectors to output more false positives. As a result, global detectors in enterprises with $\sim$100K local detectors have to process millions of alerts per day [6] which stresses heavy-weight program analyses and human analysts who investigate the

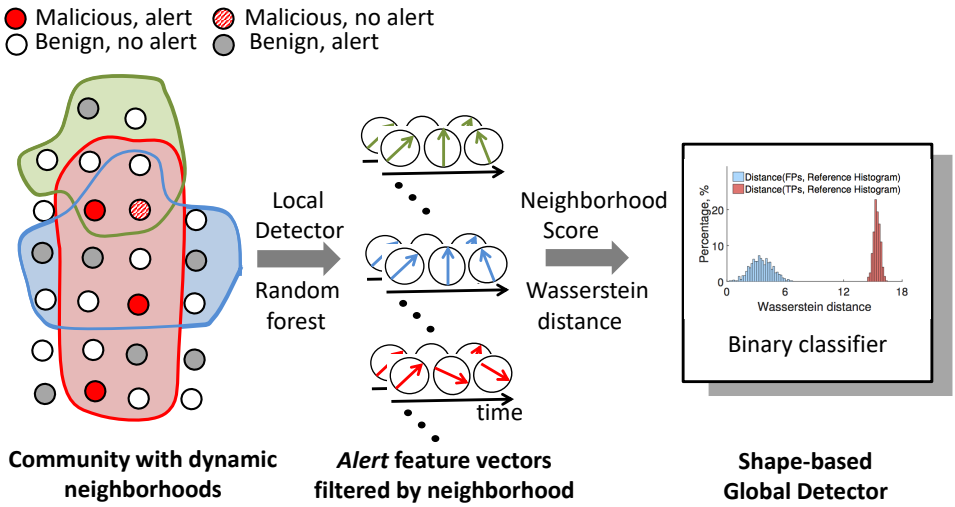

Figure 3.1: (L to R) Each circle is a node that runs a local malware detector (LD). Our goal is to create a robust global detector (GD) from weak LDs. We observe that nodes naturally form *neighborhoods* based on attributes relevant to attack vectors – e.g., all client devices that visit a website W within the last hour belong to neighborhood NB_w, or all users who received an email from a mailing list M in the last hour belong to neighborhood NB_m. We propose a new GD that groups together suspicious local feature vectors based on neighborhoods – traditional GDs only analyze local alerts while we re-analyze feature vectors that led to the alerts. Our GD then exploits a new insight – the conditional distribution of true positive feature vectors differs from false positive feature vectors – to robustly classify neighborhoods as malicious.

final alerts [18] – our goal is to build a robust global detector that amplifies weak local detectors.

Challenges for prior work. Boosting weak detectors using purely machine learning techniques is challenging. The dominant approaches are (a) clustering: combine feature vectors using some distance metric to identify suspicious clusters of feature vectors [118, 154, 160, 164], and (b) counting: train local detectors (LDs) such as Random Forests to generate local alerts, and generate a global alert if there is a significant fraction of local alerts in the enterprise [71, 88, 89, 137]. Both approaches

have limitations that force enterprises to deploy brittle rule-sets that explicitly correlate local detector alerts.

Clustering algorithms are well-known to be highly sensitive to noise, especially in the high-dimensional regime [73, 96, 155]. Indeed, classical approaches that attempt to detect or to score "outlyingness" of points (e.g. Stahel-Donoho outlyingness, Mahalanobis distance, minimum volume ellipsoid, minimum covariance determinant, etc) are fundamentally flawed in the high-dimensional regime (i.e., theoretically cannot guarantee correct detection with high probability). In practice, we see this in prior work [160] where clustering is used primarily as a first-level analysis to discover malicious incidents for a human analyst (i.e. requires lower accuracy than a global detector). In Section 3.6 we find that a clustering global detector is ineffective in early stages of infection where our detector succeeds – i.e., clustering yields an Area Under Curve (AUC) metric of only $\sim 48\%$ against waterhole attack and phishing attacks.

Count-based global detectors (Count-GD), on the other hand, suffer because they need to know the size of local detector communities extremely accurately to determine whether a significant fraction is raising alerts. In practice though, these communities of local detectors are extremely 'noisy'. For example, consider a community of machines in an enterprise who are potentially exposed to a so-called waterhole attack [49] (where a compromised webpage spreads malicious code to machines in the enterprise). Here, a malicious javascript-advertisement might be targeted by an ad-broker to only a fraction of visitors to a set of webpages. Further, the specific exploit might only succeed on a small fraction of machines that

did receive the exploit because of browser versions or patching status. Surprisingly, our experiments show that if Count-GD underestimates the number of nodes where the exploit ran successfully (i.e., the community size) by just 2%, its alerts are almost 100% false positives (similarly, overestimating the community by 14% leads to almost 0% true positives). Even small errors in estimating the number of feature vectors in the community *linearly* affects the global detector's decision thresholds.

Proposed Ideas – Neighborhood filtering and Shape. Our intuition is that a weak signal indicative of malicious behavior still separates true- and false-positive feature vectors, even though local detectors classify both as malicious. Our proposed system, Shape-GD, relies on two key insights to correctly identify malicious feature vectors.

First, attack vectors into a firewalled enterprise create short-lived and dynamic correlations across nodes – e.g., machines that visit a specific server (in watering hole attacks) or receive email from an address (in phishing attacks) are more likely to be compromised than a random machine in the community. Since an attacker cannot target a machine inside an enterprise directly, machines that have been exposed to a common attack vector have correlated alerts. We call such a set of machines a *neighborhood*. Neighborhoods thus concentrate the signal of malware activity that is otherwise not visible at the overall community level and can thus enable early detection of malware attacks. Neighborhoods are, however, extremely unpredictable and render cluster and count-based GDs ineffective – hence we propose Shape-GD to aggregate local detectors' outputs.

The second insight behind Shape-GD is that the *distributional shape of a*

set of suspicious feature vectors can robustly separate true positive neighborhoods from false positive neighborhoods. Shape-GD analyzes only those feature vectors that cause alerts by the local detectors (*alert-FVs*) instead of analyzing all feature vectors. Alert-FVs thus represent draws from one of two *conditional distributions* – i.e., distribution of malicious or benign feature vectors conditioned on being labeled as suspicious – which are similar but not the same. Next, while a single suspicious feature vector is uninformative, a set of such feature vectors can indeed be tested to come from one of two similar-but-distinct distributions. To conduct this hypothesis test, Shape-GD introduces a quantitative scoring function that maps a set of feature vectors (the alert-FVs per neighborhood) into one scalar value – the ShapeScore of the neighborhood.

Shape-GD composes the two insights – i.e., filters alert-FVs along neighborhood lines followed by computing the neighborhood's shape – and achieves two key properties: (i) the distribution of the alert-FVs strongly separates malicious and benign neighborhoods (essentially, it separates the true positive alert-FVs from false positive alert-FVs), and (ii) is robust to noise in neighborhood size estimates (i.e., we do not need accurate neighborhood sizes and only need a sufficient number of alert-FVs to make a robust hypothesis test). Specifically, Shape-GD detects malicious neighborhoods with less than 1.1% and 2% compromised nodes per neighborhood (in two case studies involving waterhole and phishing attacks respectively), at a false positive and true positive rate of 1% and 100% respectively. Neighborhood filtering and ShapeScore complement each other – neighborhoods concentrate the weak signal into a small but unpredictable set of feature vectors while ShapeScore

extracts this signal without knowing the precise number of feature vectors.

Contributions. Neighborhood filtering and shape enable structural information about attack vectors to be captured algorithmically. Our specific contributions are as follows.

- Neighborhood filtering to 'reanalyze' alert feature vectors instead of only alert time-series, and Shape-GD algorithms that exploit a new property – the statistical 'shape' of a neighborhood separates the ones with true positives from those with false positives – to classify neighborhoods as malicious or benign without knowing their size.
- An efficient detector that can identify malicious neighborhoods using only 15,000 feature vectors (roughly 15 seconds of data from a 1000-node neighborhood). The detector comprises of random forest LDs and a Shape-GD that computes ShapeScore as the Wasserstein distance between a set of alert-FVs' histograms and a reference histogram built using false positive feature vectors (created by running LDs on benign programs in uninfected machines that are used to train the GD).
- Phishing case study. Shape-GD detects a phishing attack with 1% false positive rate in a medium size enterprise network with a neighborhood of 1086 nodes when only 17.08 nodes (using temporal neighborhoods) and 4.48 nodes (with additional mailing-list based structural filtering) are infected.
- Waterhole attack case study. Shape-GD detects a waterhole attack with 1% false positive rate when only 107.5 nodes (using temporal neighborhoods)

and 139.9 (with additional server specific structural filtering) out of $\sim 550,000$ nodes are infected.

We emphasize that the LD and GD false positives (FPs) have very different interpretations. In a phishing attack, an LD FP of 1% in a neighborhood of 1000 nodes means that we will get about 10 FP alerts per second. The Shape-GD, on the other-hand, uses these LD FP alerts for decision making. Thus a GD false positive occurs when it misclassifies a neighborhood of LD alerts – a much rarer scenario.

Specifically, a GD FP rate of 1% means that in our phishing attack scenario, we will receive a global false alert about once every 100 - 300 hours. Similarly, in the waterhole scenario a global false positive occurs every 100 sec. Comparing the number of LDs' FPs that are reported to a GD in a Count GD v. Shape-GD, temporal neighborhood filtering reduces total FPs by $\sim$100$\times$ (phishing) and $\sim$200$\times$ (waterhole), while adding structural filtering reduces total FPs by $\sim$1000$\times$ and $\sim$830$\times$ respectively (Section 3.9 for details).

Finally, as an auxiliary contribution, we present a methodology to evaluate detectors where the LD and GD algorithms are tightly integrated. Existing enterprise networks provide black-box LDs (such as Blue Coat, Symantec etc) that push alert logs into 'SIEM' tools (like Splunk) where GD algorithms and visualization tools are deployed. Section 3.4 describes the limitations of three real settings we have worked on – a real enterprise dataset, a university network test-bed, and the Symantec WINE dataset. None of these allow a GD to acquire alert feature vectors from LDs. Instead, we incorporate a host-level malware analysis setup [106] into

real enterprise data center [14] and email [10] traces, vary the rates of infection systematically, and thus determine the operating range of Shape-GD agnostic of one specific sequence of events. This methodology offers a more robust measure of Shape-GD's detection rate under adaptive malware that can alter its infection behavior in response to Shape-GD's analysis.

3.2 Overview of Shape-GD

Threat model and Deployment. We assume a standard threat model where trusted local detectors (LDs) at each machine communicate with a trusted global detector (GD) that receives alerts and other metadata from the local detectors. The LDs are isolated from untrusted applications on local machines using OS- (e.g., SELinux) and hardware mechanisms (e.g., ARM TrustZone), and communicate with the enterprise's GD through an authenticated channel. The GD is trained as a standard anomaly detector – using benign data generated from uninfected (e.g. test/quality-assurance) machines that run LDs on benign software, or assuming the current state of the system as benign in order to detect future malware as anomalies.

Shape-GD fits deployment models that are common today. Currently, enterprises use SIEM tools (like HP Arcsight and Splunk) to monitor network traffic and system/application logs, malware analysis sandboxes that scan emails for malicious links and attachments, in addition to host-based malware detectors (LDs) from Symantec, McAfee, Lookout, etc. We use exactly these side-information – from network logs (client-IP, server IP, timestamp) and email monitoring tools – to instantiate neighborhoods and filter LDs' alert-FVs based on neighborhoods (Algo-

rithm 2). Upon receiving alert-FVs, Shape-GD runs its malware detection algorithm (Algorithm 5) for all neighborhoods the alert-FVs belong to. If a particular neighborhood is suspicious, then Shape-GD will notify a downstream analysis (deeper static/dynamic analyses or human analysts) and forward relevant information in the incident report.

The key difference is that Shape-GD needs to know the alert feature vectors from the LDs – black-box LDs do not currently provide these. Hence (e.g., `osquery`-based) co-designed LD-GD detectors [19, 34] are the most appropriate deployment counterpart for Shape-GD– this also motivates our experimental setup combining host-level malware analysis and web-service/email datasets.

Operationally, the LD at each machine transforms its input signal into an alert time series. This transformation consists of two steps: *(a) Generate Feature Vectors:* convert the raw OS system calls trace into a feature vector (FV) time series, and *(b) Generate Alerts:* Determine if each FV is malicious or not using a local detector (typically through random forests, SVM, etc.).

Inferring neighborhoods from common attack vectors. Shape-GD operates over dynamic neighborhoods, which are updated once per neighborhood time window (NTW). Neighborhoods within large communities are a set of nodes that share a statically defined *action attribute* within the current time window – this allows an analyst to create neighborhoods of nodes based on common attack vectors. Below are some illustrative examples of communities and neighborhoods – we focus our experiments on the first two examples that are responsible for a large fraction of malware in enterprises.

1. Waterhole attack. The community here consists of the employees of an enterprise such as Anthem Health [38, 40]. In a waterhole attack, adversaries compromise a website commonly visited by such employees as a way to infiltrate the enterprise network and then spread within the network to a privileged machine or user. Within this community, a neighborhood can be the set of nodes that visited the same type of websites within the current neighborhood time window (for example, some percentile of suspicious links rated by VirusTotal [46] or SecureRank [35]). Since these rankings themselves are fuzzy, and the websites and their contents are dynamic, neighborhoods only indicate a probability that the node was actually exposed to an exploit.

2. Phishing attack over enterprise email networks. The community here consists of all employees within an enterprise. A phishing attack here would typically spread over email and use a malicious URL to lure nodes (users) to drive-by-download attacks [26, 39] or spread through malicious attachments. Here, a specific user's neighbors are that subset of users with whom she/he exchanged emails with during the current neighborhood time window.

Similar correlations occur in physical hardware attacks – community here consists of all machines in a workplace that are physically proximal (e.g. machines in a specific hospital or bank determined using the configuration of LAN/WiFi infrastructure, GPS information etc). The potential attack mode here is through physical hardware such as `badUSB`. The neighborhood of a node is simply all other nodes in the neighborhood that were connected to similar external hardware (e.g. a USB drive) over the current neighborhood time window.

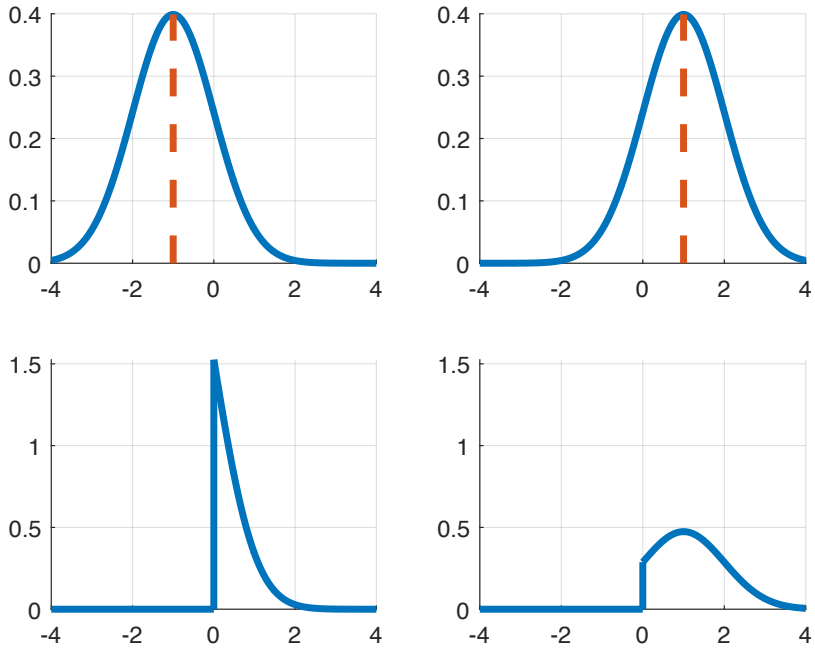

Figure 3.2: (Shape of conditional distributions) The top left figure is the probability density function (pdf) of benign feature vectors, here a Gaussian with mean '-1'; and the top right figure is the pdf for malicious feature vectors, here a Gaussian with mean'+1'. The optimal local detector at any machine would declare 'malware' if a sample's value is positive, and declare 'benign' if a sample's value is negative. The bottom plots shows the pdfs of the same Gaussians but now conditioned on the event that the sample is positive – the pdfs corresponding to false positive and true positive feature vectors respectively have different shapes.

Attacks that target specific app-stores (e.g., the Key-Raider attack in the Cydia app-store or the malicious Xcode attacks due to compromised mirror sites) also propagated across users with specific attributes (membership in a store or downloaded Xcode from specific sites) more likely than a random user in the network.

3.2.1 Intuition behind Shape-GD

The statistical shape of local detectors' false positives (FP conditional distribution) differs from the corresponding shape for true positives (TP conditional distribution) – we use this property to aggregate LDs' alert-FVs to find the shape of each neighborhood and then classify neighborhoods based on their shapes.

The central question then is – *why do true- and false-positive FVs' shapes differ?* To explain this and set the stage for Shape-GD, we consider a stylized statistical inference example. Suppose that we have an unknown number of nodes within a neighborhood. We want to distinguish between two extremes – all nodes only run benign applications (benign hypothesis), or all nodes are running malware (malware hypothesis). We look at a single snapshot of time where each node generates exactly one feature vector. Under the benign hypothesis, assume that the feature vector from each node is a (scalar valued) sample from a standard Gaussian with mean of '-1'; alternatively it is standard Gaussian with mean of '+1' under the malware hypothesis.

(a) Noisy local detectors: Given one sample (i.e., FV from one node), the *best* local detector is a threshold test: is the sample's value above zero or below? For this example, the probability of a false positive is (about) 15%.

(b) Aggregating local detectors over neighborhoods: Suppose there are 100 nodes and all of them report their value, and we are told that 90 of them are greater than 0 (i.e., 90 of the local detectors generate alerts). In this case, the expected number of alerts under the benign hypothesis is 15; and 85 under the malware

hypothesis. Thus, we can conclude with overwhelming certainty (10^{-75} chance of error) that 90 alerts indicate an infected neighborhood. This corresponds to a conventional threshold algorithm that count the number of alarms in a neighborhood and compares with a global threshold (here this threshold is 50).

(c) Count without knowing neighborhood size: Suppose, now, that we do not know the number of nodes (i.e., neighborhood size is unknown), and only know that there are a total of 90 alerts. In other words, out of the neighborhood of nodes, some 90 of them whose samples were positive reported so. What can we say? Unfortunately we cannot say much – if there were 100 nodes in neighborhood, then malware is extremely likely; however, if there were 1000 total nodes, then with 90 alerts, it is by far (exponentially) more likely that we have no infection. Because we do not know the neighborhood size, the global threshold cannot be computed.

(d) Robustness of Shape: While the number of alerts alone is uninformative, we can resolve whether the neighborhood is a 'false positive' or 'true positive' by considering the actual values of the 90 random variables corresponding to these *alerts*. These values represent independent draws from a *conditional* distribution – either the distribution of a normal random variable of mean '-1' *conditioned on taking a nonnegative value*, or the distribution of a normal random variable of mean '$+1$' *conditioned on taking a nonnegative value* (see Figure 3.2). This conditioning occurs because of the local detector – recall it tags a sample as an alert if and only if the sample drawn was nonnegative (optimal LD in this example). Thus, irrespective of the size of the neighborhood, the global detector would "look at the shape" of the empirical distribution (i.e. the distribution constructed from the received samples)

of the received samples (FVs). If this were "closer" to the left rather than the right plot in Figure 3.2, it would declare "uninfected"; otherwise declare "infected".

3.2.2 From Intuition to Algorithm Design

Interestingly, we show that the intuition behind this simple example scales to real malware detectors that use high-dimensional feature vectors. However, to use this insight in practice, we need to address two issues: *(i)* while the two figures in Figure 3.2 are visually distinct, an algorithmic approach requires a quantitative score function to separate between the (vector-valued) conditional distributions generated from feature vector samples; and *(ii)* the global detector receives only finitely many samples; thus, we can construct (at best) only a noisy estimate of the conditional distribution. We describe Shape-GD's details in Section 3.3 but present the key intuition here.

We introduce **ShapeScore** – a score function based on the Wasserstein distance [48] to resolve between conditional distributions. We choose Wasserstein distance because it has well-known robustness properties to finite-sample binning [59, 144], was more discriminative than L1/L2 distances in our experiments, and yet is efficient to compute for vectors.

Specifically, given a collection of feature vector samples, we construct an empirical (vector) histogram of the FV samples, and determine the Wasserstein distance of this histogram with respect to a *reference histogram*. This reference histogram is constructed from the feature vectors corresponding to the *false positives* of the local detectors. In other words, this reference histogram captures the statis-

tical shape of the "failures" of the LDs – i.e., those FVs that the LD classifies as malicious even though they arise from benign applications. Computing the reference histogram does *not* require analysts to manually label alert-FVs as false positives – these can be generated by running the LDs on benign software in uninfected machines (e.g., test or quality-assurance machines, by recording and replaying real user traces on benign applications on training servers, etc). Alternatively, analysts can use applications deployed currently and recompute the reference histogram periodically – this is similar to anomaly detectors where the goal is to label anomalous behaviors as (potentially) malicious.

If we had the idealized scenario of infinite number of feature vector samples, the ShapeScore would be uniquely and deterministically known. In practice however, we have only a limited number of feature vector samples; thus ShapeScore is noisy. Our experiments (Figure 3.4) test its robustness with Windows benign and malicious applications (Section 3.5), where the ShapeScore is computed from neighborhood sizes of 15,000 FVs (about 15 seconds of data from 1000 nodes). The key result is the strong statistical separation between the ShapeScores for the TP and FP feature vectors respectively, thus lending credence to our approach. Importantly, both these ideas do not depend on knowing the neighborhood size; thus they provide a new lens to study malware at a global level.

3.3 Shape GD Algorithm

Our algorithm consists of feature extraction, local detector (LD), and the global detector (GD). Our key innovations are in the GD, however, we also discuss

Algorithm 1: Local Detector

Input : Sequence of executed system calls
Output: Alert-FVs
Let *id* be LD's identifier
while *True* **do**
    syscall-hist $\leftarrow$ *r*-sec histogram of system calls
    $syscall\text{-}hist_{PCA} \leftarrow$ project *syscall-hist* on *L*-dim PCA basis
    $label \leftarrow$ BinaryClassifier($syscall\text{-}hist_{PCA}$);
    if $label = malicious$ **then**
        $alert\text{-}FV \leftarrow syscall\text{-}hist_{PCA}$
        send $< alert\text{-}FV, id >$ to Shape-GD

feature extraction and LD design for completeness.

Local Detector (LD) Algorithm. Our first step is to establish a good local detector (LD) for desktop systems running Windows OS. In particular, we choose system call based LDs since the system call interface has visibility into an app's intercation with core OS components – file system, Windows registry, network – and can thus capture signals relevant to malware executions.

We experiment with an extensive set of system-call LDs – our takeaway is that even the best LD we could construct operates at a true- and false-positive ratio of 92.4%:6% and is not deployable by itself (i.e., will create $\sim$30 false positives every 10 minutes without a GD).

Each LD comprises of a feature extraction (FE) algorithm and a machine learning (ML) classifier. Our FE algorithm partitions the time-series of system calls into 1-sec chunks and represents each chunk as a histogram (where each bin contains frequency of a particular system call). Then it projects all feature vectors onto 10-dimensional subspace spanned by top 10 principal components generated

by PCA algorithm. We choose ML classifiers (used throughout prior work because these are computationally efficient to train) such as SVMs, random forest, k-Nearest Neighbors, etc, and do not include complex alternatives such as artificial neural networks or deep learning algorithms. We also deliberately avoid handcrafted ML algorithms and hardcoded detection rules.

Figure 3.3 plots the true positive v. false positive rates (i.e. the ROC curves) of the seven ML algorithms we evaluate. The area under the ROC curve (AUC) is a quantitative measure of LD's performance: the larger the AUC, the more accurate the detector. We specifically experiment with seven state-of-the-art ML algorithms: random forest, 2-class SVM, kNN, decision trees, naive Bayes, and their ensemble versions – boosted decision trees with AdaBoost algorithm and Random SubSpace ensemble of kNN classifiers (Figure 3.3). We also evaluated 1-class SVM as an anomaly detector – however, it yielded an extremely high FP rate and we exclude it from further discussion. Overall, the random forest classifier worked best – it has the highest AUC and we pick an operating point of 92.4% true positives at a false positive rate of 6%.

Neighborhood Instances from Attack-Templates. Each neighborhood time window (NTW), Shape GD generates neighborhood instances (Algorithm 2) based on statically defined attack vectors – each attack vector is a "Template" to generate neighborhoods with. Algorithm 2 shows how the concept of neighborhood unifies operationally distinct attacks like waterhole and phishing.

The template for detecting a waterhole attack forms a neighborhood out of client nodes that access a server or a group of servers within a neighborhood

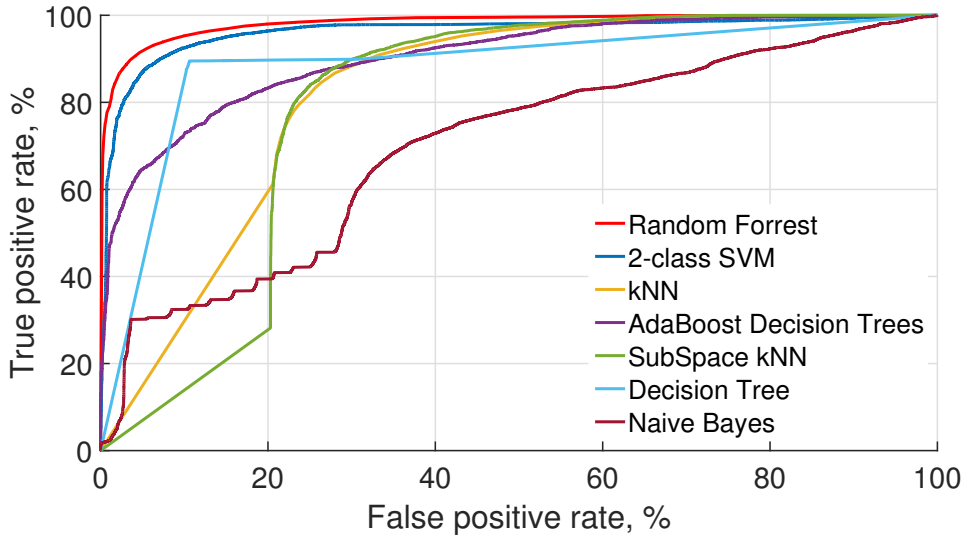

Figure 3.3: (ROC curves) True positive v. False positive curves shows detection accuracy of seven local detectors. Random Forest outperforms all others; but has unacceptably high false positive rate (above 10%) if one wants to achieve at least 95% true positive rate.

time window. The other template, which is used for detection of a phishing attack, includes in a neighborhood email recipient machines belonging to a set of mailing lists. The two templates are shown in lines 3–10.

For simplicity we present a batch version of the neighborhood instantiation algorithm (Algorithm 2) which advances time by NTW and creates new neighborhoods for each NTW. In contrast, the online Shape GD version updates already existing neighborhoods while monitoring client–server interactions in real-time – we demonstrate the online Shape GD algorithm to detect waterhole attacks and the batch version against phishing attacks in our evaluation.

The neighborhood instantiation algorithm accepts a template type as input, i.e. either a template for detecting a waterhole attack or a phishing attack, and

Algorithm 2: Neighborhoods from Attack-Vectors

Input : Template-type, NTW
Output: Set of active neighborhoods NBDs

[time, time+NTW] defines the current time window

time ← *current time*

while *True* **do**
  if *Template-type* = *waterhole attack* **then**
    V := client machines*
    S := accessed servers*
    predicate(A:Client, B:Servers) := A accesses B
  else if *Template-type* = *phishing attack* **then**
    V := email recipient machines*
    S := mailing lists*
    predicate(A:Recipient, B:Mailing list) := $A \subseteq B$
  partitioning a set into non-disjoint sets to incorporate structural filtering
  $P_1, P_2, ..., P_N \leftarrow$ partition-set(S), where $S = \bigcup_{i=1}^{N} P_i$
  form neighborhoods NB_i using partitions P_i
  $NB_i \leftarrow \{V \mid \text{predicate}(V, P_i)\}$
  set expiration time for a neighborhood NB_i
  NB_i.expiration-time ← *t+NTW*
  add all neighborhoods to the set NBDs
  $NBDs \leftarrow \{NB_i \mid \forall i \text{ in } [1, N]\}$
  advance time by NTW sec
  time ← *time+NTW*

**active within the time window [time, time+NTW]*

duration of an NTW. The algorithm runs once per NTW – starting by defining the sets V and S that will be used to form neighborhoods. For a waterhole attack, the set V includes all client machines accessing a set of servers and S is a set of the accessed servers. To instantiate neighborhoods for a phishing attack, V is a set of all email recipient machines and S is a set of mailing lists. In both cases the algorithm considers only the entities that are active within a current NTW window.

Each attack requires a predicate that determines relation between the elements of the sets V and S. For a waterhole attack such a predicate is true if a client *accesses* one of the servers (line 6). In the case of a phishing template, the predicate is evaluated to true if a recipient *belongs* to a particular mailing list (line 10).

The neighborhood instantiation algorithm proceeds with partitioning the set S into one or more disjoint subsets P_i (line 11). This is to incorporate 'structural filtering' into the algorithm, allowing an analyst to create neighborhoods based on subsets of servers (instead of all servers in case of waterhole) or divide all mailing lists into subsets of mailing lists (in the phishing). Structural filtering boosts detection under certain conditions (see Section 3.5.3).

The neighborhood instantiation algorithm builds a neighborhood for each partition P_i using a corresponding predicate (line 12). After forming a neighborhood, the algorithm sets its expiration time (line 13), which is the end of the current NTW window. All the neighborhoods in the set *NBDs* are discarded at the end of the current NTW window. Finally, the algorithm adds the just formed neighborhoods to *NBDs* (line 14) and advances time by one NTW (line 15).

The template-based neighborhood instantiation algorithm (Algorithms 2) shares the *NBDs* data structure with the Algorithm 5 that uses neighborhoods' shapes to detect malware.

Malware Detection in a Neighborhood. Algorithm 5 detects malware *per neighborhood* instead of individual nodes. The input to the algorithm is a set of alert-FVs from each neighborhood and its output is a global alert for the neighborhood. We now describe how the algorithm distinguishes between the *conditional distributions* of alert-FVs from true-positive and false positive neighborhoods.

The key algorithmic idea is to first extract neighborhood-level features – i.e., to map all alert-FVs within a neighborhood to a *single* vector-histogram which robustly captures the neighborhood's statistical properties. Then, Shape GD compares this vector-histogram to a reference vector-histogram (built offline during training) to yield the neighborhood's ShapeScore. The reference vector-histogram is constructed from a set of false positive alert-FVs – thus, it captures the statistical shape of misclassifications (FPs) by the LDs but at a neighborhood scale. Finally, Shape-GD trains a classifier to detect anomalous ShapeScores as malware. This is a key step in Shape-GD – i.e., mapping alert-FVs from a neighborhood into a *single* vector-histogram and then into a discriminative yet robust ShapeScore lets us analyze the joint properties of all alert-FVs generated within a neighborhood without requiring the FVs to be clustered or alerts to be counted. We describe these steps in further detail.

Generating histograms from alert-FVs. The algorithm aggregates L-dimensional projections of alert-FVs on per neighborhood basis into a set B (Algorithm 5, line

Algorithm 3: Malware Detection in a Neighborhood

Input : L-dim projections of alert-FVs
Output: Malicious neighborhoods

1 Let *NBDs* be a set of neighborhoods
2 **for** *each NB in NBDs* **do**
    aggregate L-dim projections of alert-FVs on per neighborhood basis
3 $B \leftarrow \{alert - FVs \mid \text{node } id \subseteq NB\}$
    build an (L,b)-dim. vector-histogram
4 $H_B \leftarrow$ bin & normalize B along each dimension
    compute a neighborhood score – ShapeScore
5 $ShapeScore \leftarrow$ Wasserstein Dist.(H_B, H_{ref})
    perform hypothesis testing
6 **if** $ShapeScore > \gamma$ **then**
7 label *NB* as *malicious*

3). After that, Shape GD converts low dimensional representation of alert-FVs, the set B, into a single (L,b)-dimensional vector-histogram denoted by H_B (line 4). The conversion is performed by binning L-dimensional vectors within the B set along each dimension. In each of the L-dimensions, the scalar-histogram of the corresponding component of the vectors is binned and normalized. Effectively, a vector-histogram is a matrix Lxb, where L is the dimensionality of alert-FVs and b is the number of bins per dimension.

We use standard methods to determine the size and number of bins and note that the choice of Wasserstein distance in the next step makes Shape GD robust against variations due to binning. In particular, we tried square-root choice, Rice rule, and Doane's formula [8] to estimate the number of bins, and we found that 20–100 bins yielded separable histograms (as in Figure 3.4) for the Windows dataset and fixed it at 50 for our experiments.

ShapeScore. We get the ShapeScore by comparing this histogram, H_B, to a *reference histogram*, H_{ref}, which is generated during the training phase using only the false positive FVs of the LDs. We run LDs on the system-call traces generated by benign apps – the FVs corresponding to the alerts from the LD (i.e., the false positives) are then used to construct the reference histogram H_{ref}. ShapeScore is thus the distance of a neighborhood from a benign reference histogram – a high score indicates potential malware.

To collect known benign traces, a straightforward approach is to use test inputs on benign apps or use record-and-replay tools to re-run real user inputs in a malware-free system. Or, like any anomaly detector, an enterprise can train ShapeGD using applications deployed currently and recompute H_{ref} periodically.

The ShapeScore of the accumulated set of FVs, B, is given by the sum of the coordinate-wise Wasserstein distances [144] (Algorithm 5, line 5) between

$$H_B = (H_B(1)\ H_B(2)\ \ldots\ H_B(L))$$

and

$$H_{\text{ref}} = (H_{\text{ref}}(1)\ H_{\text{ref}}(2)\ \ldots\ H_{\text{ref}}(L)).$$

In other words,

$$\text{ShapeScore} = \sum_{l=1}^{L} d_W(H_B(l), H_{\text{ref}}(l)),$$

where for two scalar distributions p, q, the Wasserstein distance [48, 144] is given by

$$d_W(p,q) = \sum_{i=1}^{b} \left| \sum_{j=1}^{i} (p(j) - q(j)) \right|.$$

This Wasserstein distance serves as an efficiently computable one dimensional projection, that gives us a discriminatively powerful metric of distance [59, 144]. Because the Wasserstein distance computes a metric between distributions – for us, histograms normalized to have total area equal to 1 – it is invariant to the number of samples that make up each histogram. Thus, unlike count-based algorithms, *it is robust to estimation errors in community size*. Figure 3.4 verifies this intuition, and shows that true positives and false positive feature vectors separate well when viewed through the ShapeScore.

Finally, to determine whether a neighborhood has malware present we perform hypothesis testing. If ShapeScore is greater than a threshold γ, we declare a global alert, i.e., the algorithm predicts that there is malware in the neighborhood (lines 6–7). The robustness threshold γ is computed via standard confidence interval or cross-validation methods with multiple sets of false-positive FVs (see Section 3.5.1).

Computing Shape GD's parameters. Here we elaborate on the steps that should be taken in a real world environment to choose parameters. The steps discussed here are generic and are applicable to other attacks beyond waterhole and phishing – the following results section quantifies each of these steps.

First, an analyst should start with designing an appropriate algorithm to run on local detectors (LDs). To achieve this, an analyst needs to compare the performance of multiple feature extraction (FE) algorithms combined with a diverse set of machine learning classifiers. One way to choose the best pair of a FE algorithm and a classifier is to build ROC curves for each pair, and select the pair that meets

the desired detection rate to computation/training effort for the LD.

Second, the analyst needs to determine whether even a purely malicious neighborhood can be separated from benign ones, and the minimum number of FVs per neighborhood to do so (Sections 3.5.1 and Section 3.7). This number depends on the false positive rate of LDs (e.g., in our experiments, we determined that a neighborhood should generate at least 15K FVs, see Figure 3.14).

Third, we need to choose an NTW based on the false positive rate (FP) and the desired time-to-detection (Section 3.5.2. A small NTW means more frequent transfers of FVs from LD to GD, whereas a long NTW means that more nodes can get compromised before the GD makes a decision and/or FPs can drown out TPs. Similarly, structural filtering can improve detection rate if the true positive alert-FVs are not deluged by the rate of false-positive alert-FVs – Section 3.5.3 quantifies how this trade-off differs for waterhole and phishing.

3.4 Experimental Setup

3.4.1 Case for a New Methodology

Shape-GD experiments require datasets where the global detector can acquire alert-FVs from local detectors, similar to `osquery`-based systems where the LD and GD are co-designed. We describe our experience with three existing methodologies and datasets – none of them allow alert-FVs to be acquired, provide complete ground truth infection information, or allow the infection rate to be varied. This motivates the methodology we use to systematically evaluate Shape-GD.

Analysis of existing datasets. Prior work has used enterprise logs [120] that are unavailable publicly. We have acquired similar security logs from a Fortune 500 company with a 200K machine network – the logs average 250M entries per day over a 2 year period, arise from 20 closed-source endpoint local detectors such as Symantec, McAfee, Blue Coat, etc, have almost 500 sparsely populated dimensions per log entry, and about 75% of the log entries lack important identity and event-timestamp information and are delayed by up to 60 days. Commercial (black-box) LDs do not expose feature vectors for external analysis.

We have also acquired network `pcap` traces from our university network, emulating prior work [90] in network-only global detectors. University security groups (like ours) are only allowed to collect network-layer `pcap` information for a rolling 2-week period and cannot instrument host machines (that are owned by students and visitors) – i.e., our 4TB/day dataset from 150K machines is unsuitable to evaluate Shape-GD because it doesn't have LDs. Extending this dataset with a weak LD – the ability to inspect executables in a sandbox downloaded by hosts (e.g., as pursued by Lastline [25]) – would be an appropriate experimental setup but sensitive data issues make such datasets hard to get. Hence, we model this extended setup in our methodology.

We have analyzed Symantec's WINE dataset [110] and found it inadequate to evaluate Shape-GD even after layering VirusTotal information on it. Specifically, the WINE dataset includes downloader graphs [110] – the nodes are executables and the edges represent whether the source downloaded the destination executable – and represent downloader trojans ('droppers') in malware distribution networks [130]

that download payloads to steal information, encrypt the disk, etc on to the host. This dataset covers 5 years of data with 25M files (specifically, hashes that represent files) on over 1M machines – however, only 1.5M of the 25M hashes have reports on VirusTotal. Hence, one cannot reconstruct (alert) feature vectors for the hashes stored in the downloader graph.

Shape-GD re-analyzes (local) alert feature vectors in the global detector – filtering alert-FVs into neighborhoods and then computing the neighborhoods' shapes. Hence we model `osquery`-like deployments as used in enterprises like Facebook and Google where the LDs and GDs are co-designed and GDs can acquire alert-FVs.

Simulating malware propagation in a network. Methodologies that use existing datasets with malware propagation (like the ones above) have an inherent weakness. Such datasets have one sequence of malware propagation events "hardwired" into the dataset and do not allow us to analyze how a detection mechanism reacts to variations of malware propagation dynamics, especially when malware can adapt these dynamics. Instead, we propose to vary the rate of infection (which changes the neighborhood formation) and determine Shape-GD's detection performance across different infection rates.

Further, none of the above datasets provide ground truth information about the true extent of infections, incentivizing a design that minimizes false positives at the expense of false negatives. In a controlled setting where host-level malware and benignware traces are overlaid onto a trace of web-service/email communication, we can maintain ground truth information and determine false positives and

negatives precisely.

To this end, we use a malware and benignware dataset from a recent related work [106], train an LD with histogram-based feature vectors and a Random Forest detector based on a recent survey on host-level malware detection [62], and overlay these host-level results on web-service (network) and email traces using two standard (and publicly available) datasets from Yahoo data centers and Enron respectively. We describe this methodology in detail next.

3.4.2 Benign and Malware Applications

We collect data from thousands of benign applications and malware samples. To avoid tracing program executions where malware may not have executed any stage of its exploit or payload correctly, we set a threshold of 100 system calls per execution to be considered a success. Our experiments successfully run 1,311 malware samples from 193 malware families collected in July 2013 [106], and 2,364 more recent samples from 13 popular malware families collected in 2015 [24], to compare against traces from 1,889 benign applications.

We record time stamped sequences of executed system calls using Intel's Pin dynamic binary instrumentation tool. Each Amazon AWS virtual machine instance runs Windows Server 2008 R2 Base on the default T2 micro instances with 1GB RAM, 1 vCPU, and 50GB local storage. The VMs are populated with user data commonly found on a real host including PDFs, Word documents, photos, Firefox browser history, Thunderbird calendar entries and contacts, and social network credentials. To avoid interference between malware samples, we execute each

sample in a fresh install of the reference VM. As malware may try to propagate over the local network, we set up a sub-net of VMs accessible from the VM that runs the malware sample. In this sub-net, we left open common ports (HTTP, HTTPS, SMTP, DNS, Telnet, and IRC) used by malware to execute its payload. We run each benign and malware program 10 times for 5 minutes per run for a total of almost 53,000 hours total compute time on Amazon AWS.

Overall, benignware and malware were active for 141,670 sec and 283,270 seconds respectively, executing an average of 11,900 and 13,500 system calls per second respectively. Using 1 second time window (Section 3.3) and sliding the time windows by 1ms, we extract histograms of system calls within each time window as the ML feature, and finally pick 1.5M benign and 1M malicious FVs from this dataset for the experiments that follow. Importantly, we do not constrain the samples on neighboring machines to belong to the same families – as described above, malware today predominantly spreads through malware distribution networks where a downloader trojan ('dropper') can distribute arbitrary and unrelated payloads on hosts. We want to test Shape-GD in the extreme case where malicious FVs can be assigned from any malware execution to any machine.

3.4.3 Modeling Waterhole and Phishing Attacks

Waterhole attack. To model a waterhole attack, we use Yahoo's "G4: Network Flows Data" [14] dataset, which contains communication data between end-users and Yahoo servers. The 41.4 GB (in compressed form) of data were collected on April 29-30, 2008. Each netflow record includes a timestamp, source/destination

IP address, source/destination port, protocol, number of packets and the number of bytes transferred from the source to the destination. We model the setting where heuristics such as SecureRank [35] are applied to identify suspicious servers and we assume that Shape-GD monitors the top (here, 50) suspicious servers based on SecureRank's scores. Specifically, we use 5 hours of network traffic (208 million records) captured on April 29, 2008 between 8 am and 1 pm at the border routers connecting Dallas Yahoo data center (DAX) to the Internet. The selected 50 DAX servers communicate with 3,181,127 client machines over 14,249,931 requests.

We assume that an attacker compromises one of the most frequently accessed DAX server – 118.242.107.76, which processes $\sim 752,000$ requests within 5-hour time window (~ 43.7 requests per second). In our simulation it gets compromised at random instant between 8am and 10.30am. Hence, Shape GD can use the remaining 2.5 hours to detect the attack (our results show that less than a hundred seconds suffice). Following infection, we simulate this 'waterhole' server compromising client machines over time with an infection probability parameter – this helps us determine the time to detection at different rates of infection. The benign and compromised machines then select corresponding type of execution trace (i.e., a sequence of FVs generated in Section 3.4.2) and input these to their LDs.

Phishing attack. We simulate a phishing attack in a medium size corporate network of 1086 nodes that exchange emails with others in the network. To model email communication, we pick 50 email threads with 100 recipients each from the publicly available Enron email dataset [10] (the union of all email threads' recipients is 1086).

We start the simulation with these 50 emails being sent into the 1086-node neighborhood, and seed only *one* email out of 50 as malicious. We then model the infection speading at different rates as this malicious email is opened by its (up to 100) recipients at some time into the simulation and is compromised with some likelihood when the user 'clicks' on the URL in the email. Our goal is to measure the number of compromised nodes before Shape GD declares an infection in this neighborhood. All nodes that open and 'click' the link in the malicious email will select malware FVs from Section 3.4.2 as input to their corresponding LDs, while the remaining nodes select benign FVs.

To simulate the infection spreading over the email network, we need to (a) model when a recipient 'opens' the email: we do so using a long tail distribution of reply times where the median open time is 47 minutes, 90-percentile is one day, and the most likely open time is 2 minutes [108]; and (b) model the 'click' rate (probability that a recipient clicks on a URL): we vary it from 0% up to 100% to control the rate of infection. For example, within 1-, 2-, 3-hour long time interval only 55%, 65%, and 70% of recipients of a malicious email open it, which corresponds to 55, 65, and 75 infected machines respectively at 100% click rate.

Overall, these two scenarios differ in their time-scales (seconds v. hours) and in the relative rate at which benign and malicious neighborhoods grow. As we will see, these parameters have a significant impact on the composition of neighborhoods and the Shape GD's detection rate.

Methodology. We report averaged results from repeating each experiment multiple times with random initialization parameters. In particular, we use 10-fold

cross validation for machine learning experiments (Figure 3.3), 500 randomly sampled benign/malicious neighborhoods with 10 repetitions to compute average (Figures 3.4, 3.14), 100 repetitions of each malware infection experiment (Figures 3.5,3.6,3.12,3.11), and 100 repetitions of infection with 10 repetitions per datapoint (Figures 3.7,3.8,3.9,3.10). To train the reference histogram, H_{ref}, we select 15K FVs and 100K FVs from the training data set in the phishing and waterhole experiments respectively. All Shape GD's parameters are chosen based on a training data set (used for Figures 3.4 and 3.14) – we then evaluate Shape GD (in the remaining figures) using a completely separate testing data set.

3.5 Results

We show that Shape-GD can identify malicious neighborhoods with *less than 1% false positive and 100% true positive rate when the neighborhoods produce more than 15,000 FVs within a neighborhood time window (i.e.,* $|B| > 15,000$ *in Algorithm 5*). Recall that at 60 FVs/node/minute, it takes 1000 nodes only 15 seconds to create 15,000 FVs. For LDs like ours with $\sim$6% false positive rate, this corresponds to 900 alert-FVs. We then simulate realistic attack scenarios and find that Shape-GD can detect malware when only 5 of 1086 nodes are infected through phishing in an enterprise email network, and when only 108 of 550K possible nodes are infected through a waterhole attack using a popular web-service. Finally, Shape-GD is computationally efficient – we relegate this discussion to Section 3.8.

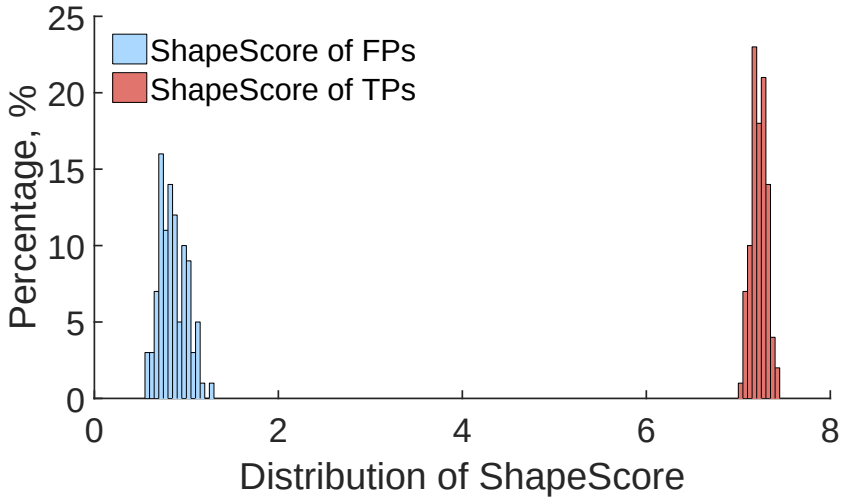

Figure 3.4: Histogram of the ShapeScore: The ShapeScore is computed for neighborhoods with 15,000 FVs each (experiment repeated 500 times to generate the histograms). Shape-based GD can reliably separate FPs and TPs through extracting information from the data that has been unutilized by an LD.

3.5.1 Can Shape of Alert-FVs Identify Malicious Neighborhoods?

We first show that the shape of a neighborhood can easily distinguish between neighborhoods that are either 100% benign or 100% malicious. We quantify Shape-GD's time to detection under real settings with a mix of both in subsequent sections.

Figure 3.4 shows that Shape-GD can indeed separate purely benign neighborhoods from purely malicious ones. To conduct this experiment, we construct purely benign and malicious neighborhoods with $\sim$15,000 benign or malicious FVs respectively (i.e., $|B|$ is 15,000). In Section 3.7, we experimentally quantify the sensitivity of Shape-GD to the number of FVs in a neighborhood ($|B|$) and find that neighborhoods with more than 15,000 FVs lead to robust global classification.

For each neighborhood, we use the Random Forest LD to generate *alert-FVs* and use Shape-GD to compute the neighborhood's ShapeScore using the alert-FVs from the neighborhood. In Figure 3.4, we plot histogram of ShapeScores for 500 benign and malicious FVs each – each point in the blue (or red) histogram represents the ShapeScore of a completely benign (or malicious) neighborhood. Recall that a small ShapeScore indicates the neighborhood's statistical shape is similar to that of a benign one. *The non-overlapping distributions separated by a large gap indicate that the shape of purely benign neighborhoods is very different from the shape of purely malicious neighborhoods.*

Shape-GD detects anomalous neighborhoods by setting a threshold score based on the distribution of benign neighborhoods' scores (Figure 3.4) – if an incoming neighborhood has a score above the threshold, Shape GD labels it as 'malicious', otherwise 'benign'. We set the threshold score at 99-percentile (i.e. our expected *global false positive rate* is 1%) and the true positive rate is effectively 100% for this experiment. This shows that for homogeneous neighborhoods producing over 15K FVs within a neighborhood time window, Shape-GD can make robust predictions. The next question then is how well Shape-GD can do so when neighborhoods are partially infected – we evaluate this in the next section.

3.5.2 Time to Detection Using Temporal Neighborhoods

Temporal filtering creates a neighborhood using only the nodes that are *active* within a neighborhood time window (NTW). For example, a temporal neighborhood for the phishing scenario would include every email address that received

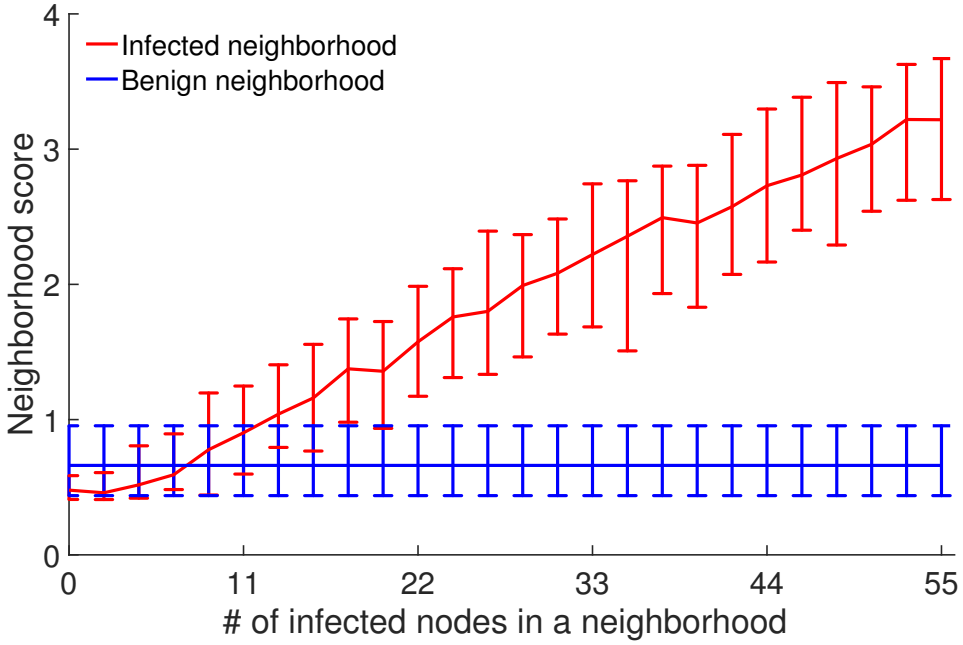

Figure 3.5: (Phishing attack: Time-based NF) Dynamics of an attack: While the portion of infected nodes in a neighborhood increases over time reaching 55 nodes out of 1086 on average, ShapeScore goes up showing that Shape GD becomes more confident in labeling neighborhoods as 'malicious'. It starts detecting malware with at most 1% false positive rate when it compromises roughly 22 nodes. The neighborhood includes all 1086 nodes in a network and spans over 1 hour time interval.

an email within the last hour (1086 nodes in our experiments). Similarly, a waterhole attack scenario would include all client devices that accessed *any* server within the last NTW into one neighborhood ($\sim 17,000$ nodes on average in 30 seconds). This neighborhood filtering models a CIDS designed to detect malware whose infection exhibits temporal locality (and obviously does not detect attacks that target a few high-value nodes through temporally uncorrelated vectors).

Phishing and waterhole attacks operate at different time scales (and hence NTWs). Due to the long tail distribution of email 'open' times, the phishing NTW

varies from 1–3 hours in our experiments. On the other hand, a popular waterhole server quickly infects a large number of clients within a short period of time – thus, we vary the waterhole NTW from 4 seconds up to 100 seconds.

Shape GD's time to detection for one NTW. We fix NTWs (1 hour for phishing and 30 seconds for waterhole) and vary a parameter that represents a node's likelihood of infection from 0% up to 100% – modeling whether a phished user clicks the malicious URL (phishing) or a drive-by exploit succeeeds in a waterhole attacks.

Figures 3.5 and 3.6 plot the neighborhood score v. the average number of infected nodes within benign (blue curve) and malicious (red curve) neighborhoods – the two extreme points on the X-axis corresponds to either none of the machines being infected (the left side of a figure) or the maximum possible number of machines being infected (the right side of the figure). In this experiment, phishing can infect up to 55 machines in the 1 hour NTW, while the waterhole server can infect almost 1250 nodes in the 30 seconds NTW. Every point on a line is the median neighborhood score from 10 experiments with whiskers set at 1%- and 99%-percentile scores.

When increasing the number of infected nodes in a neighborhood, as expected, the red curve larger deviates from the blue one. Therefore, Shape GD becomes more confident with labeling incoming partially infected neighborhoods as malicious. Shape GD starts reliably detecting malware very quickly – when only 22 nodes (phishing) and 200 nodes (waterhole) have been infected. We also experimented with other sizes of neighborhood window – the plots we obtained showed similar trends.

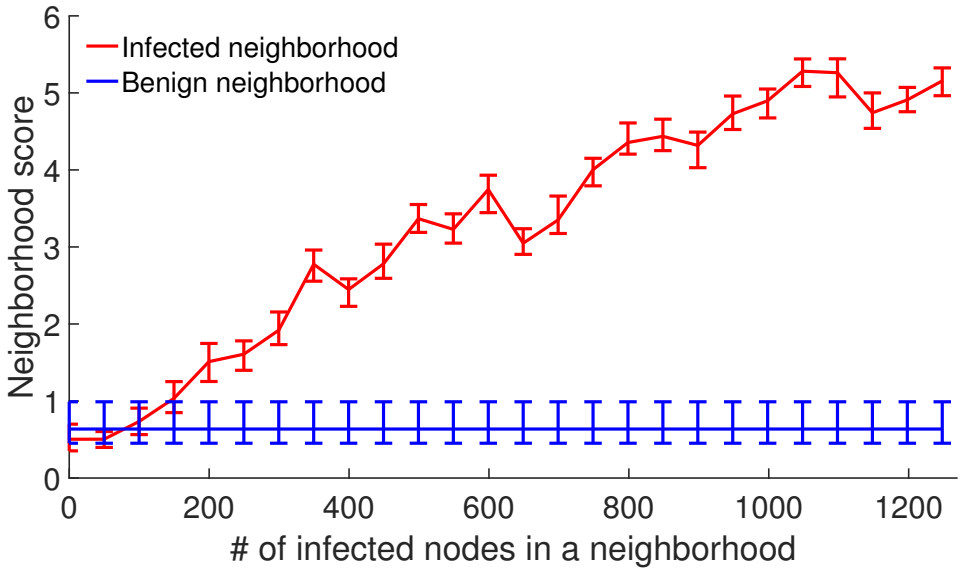

Figure 3.6: (Waterhole attack: Time-based NF) Dynamics of an attack: While the portion of infected nodes in a neighborhood increases over time reaching 1248 nodes on average, ShapeScore goes up showing that Shape GD becomes more confident in labeling neighborhoods as 'malicious'. It starts detecting malware with at most 1% false positive rate when roughly 200 nodes get compromised. The neighborhood includes 17,178 nodes on average and spans over 30 sec time interval.

Shape GD's sensitivity to NTW. We show that the size of a neighborhood is important for early detection – the minimum number of nodes that are infected before Shape GD raises an alert – in Figures 3.7 and 3.8. Varying the NTW essentially competes the rates at which both malicious and benign FVs accumulate – interestingly, we find that *these relative rates are different for phishing and waterhole attacks and lead to different trends for detection performance v. NTW*.

We vary the NTW from 1 hour to 3 hours for phishing and from 4 sec to 100 sec for waterhole and record the number of infected nodes when Shape GD can make robust predictions (i.e. less than 1% FP for almost 100% TP).

Increasing the NTW in the phishing experiment from 1 to 3 hours *improves* the Shape GD's detection performance – at 17.08 infected nodes for a 3 hour NTW compared to 20.24 nodes for a 1 hour NTW. Detection improves slowly because while the infection rate slows down over time as fewer emails remain to be opened, the long tail distribution of email 'open' times causes most of the 17 victims to fall early in the NTW and accumulate sufficient malicious FVs to tip the overall neighborhood's shape into malicious category.

In a waterhole scenario, the number of client devices active within a time window (and hence the false positive alert-FVs from the neighborhood) grows much faster than the malware can spread (even if we assume that *every* client that visits the waterhole server gets infected. Here, a large NTW aggregates many more benign (false positive) FVs from clients accessing non-compromised servers. Hence, in contrast to the phishing attack, increasing the NTW degrades time to detection. Shape GD works best with an NTW of 6 seconds – only 107.5 nodes on average become infected out of a possible $\sim$550,000 nodes. Note that a very small NTW (below 6 seconds) either does not accumulate enough FVs for analysis – if so, Shape GD outputs no results – or creates large variance in the shape of benign neighborhoods and abruptly degrades detection performance.

Note that a Shape GD requires a minimum number of FVs per neighborhood to make robust predictions – at least 15,000 FVs based on Section 3.7 – hence, the Shape GD has to set NTWs based on the rate of incoming requests and access frequency of a particular server. For example, if a server is not very popular and is likely to be compromised, the Shape GD could increase this server's NTW to

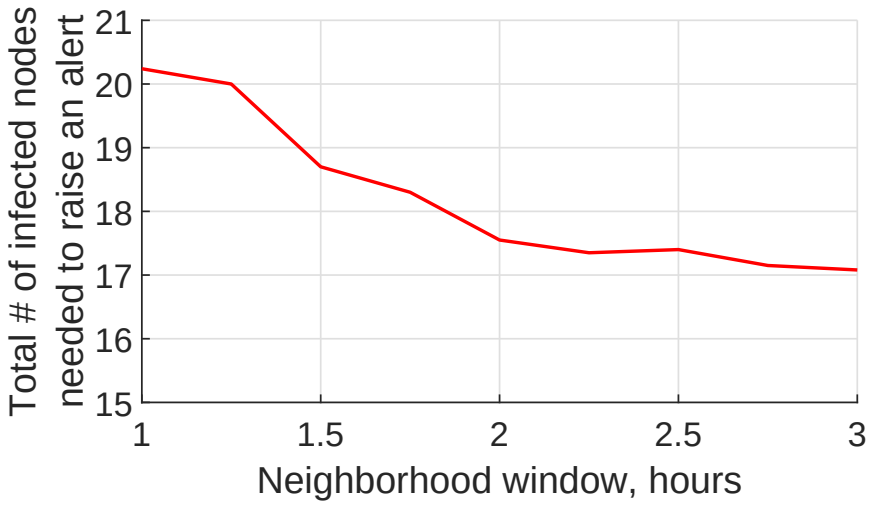

Figure 3.7: (Phishing attack: Time-based NF algorithm) Shape GD's performance improves by 18.5% (20.24 and 17.08 infected nodes) when increasing the size of a neighborhood window from 1 hour to 3 hours.

collect more FVs for its neighborhood.

3.5.3 Time to Detection Using Structural Information

Both phishing and waterhole attacks impose a logical structure on nodes (beyond their time of infection): phishing spreads malware through malicious email attachments or links while waterhole attacks infect only the clients that access a compromised server. This structure suggests that temporal neighborhoods can be further refined based on the sender/recipient-list of an email (e.g., grouping members of a mailing list into a neighborhood in the phishing scenario) or based on the specific server accessed by a client (i.e., grouping clients that visit a server into one neighborhood).

To analyze the effect of such structural filtering on GD's performance, we

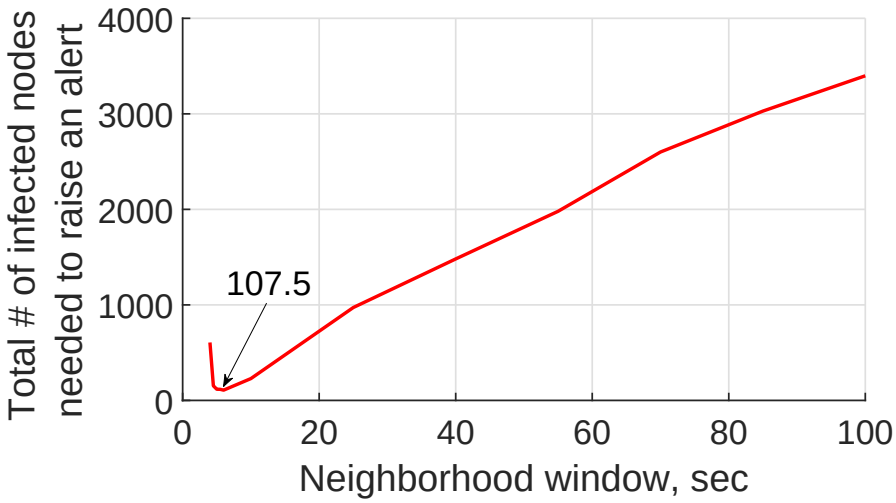

Figure 3.8: (Waterhole attack: Time-based NF) Shape GD's performance deteriorates linearly when increasing the size of a neighborhood window from 6 sec to 100 sec.

vary filtering from coarse- (no structural filtering, only time-based filtering) to fine-grained (aggregating alerts across each recipients' list separately or across clients accessing each server separately) (Figures 3.9, 3.10). Specifically, the aggregation parameter changes from 50 recipients' lists or servers down to 1. As before, we measure detection in terms of the minimum number of infected nodes that lead to raising a global alert. Also we consider three NTW values – 1-, 2-, and 3- hours long for phishing and 25-, 50-, and 100-sec long for waterhole.

Figure 3.9 shows that structural filtering improves detection of a phishing attack by $\sim 4x$ (difference between left and right end points of each curve) over temporal filtering – by filtering out alert-FVs from unrelated benign nodes that were active during the same NTW as infected nodes. Interestingly, the size of a neighborhood window does not considerably affect the detection when used along with

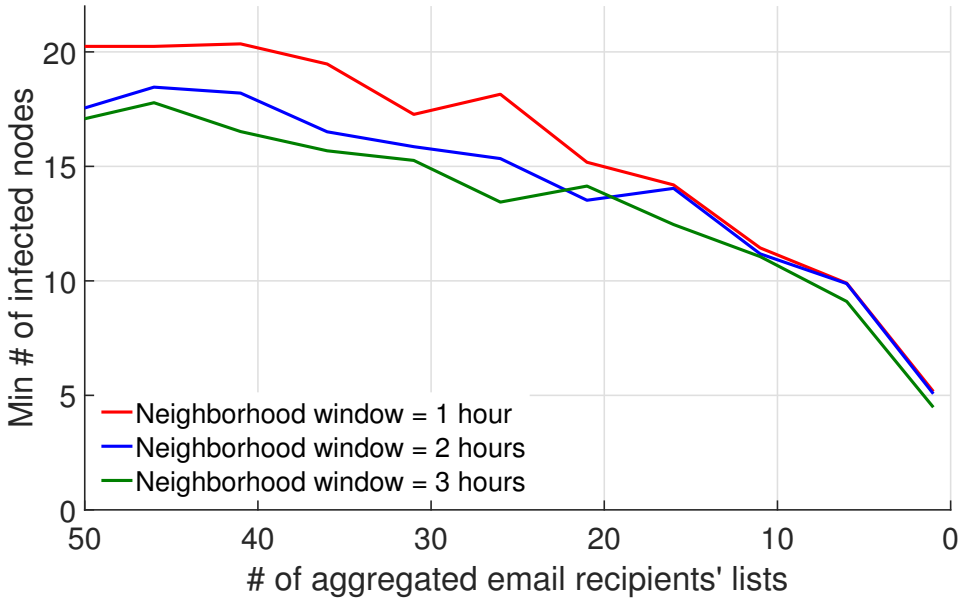

Figure 3.9: (Phishing attack) Comparing to pure time-based NF, structural filtering algorithm improves Shape GD's performance by $\sim 4\times$ by taking into consideration logical structure of electronic communication (sender – receiver relation).

the most fine-grained structural filtering (treating each recipients' list individually) – 3-hour long NTWs results in only a $\sim$ 12% decrease in the number of compromised nodes (i.e. time to detection). This shows that there is substantial signal that structural filtering can help extract from alert-FVs in smaller NTWs (and thus improve Shape GD's time to detection).

Structural filtering improves time to detect waterhole attacks as well – by 5.82x, 4.07x, and 3.75x for 25-, 50-, 100-sec long windows respectively. Interestingly, structural filtering requires Shape GD to use longer NTWs than before – small NTWs (such as 6 seconds from the last sub-section) no longer supply a sufficient number of alert-FVs for Shape GD to operate robustly. Even though structural filtering with a 25 second NTW improves detection by 5.82x over temporal filtering

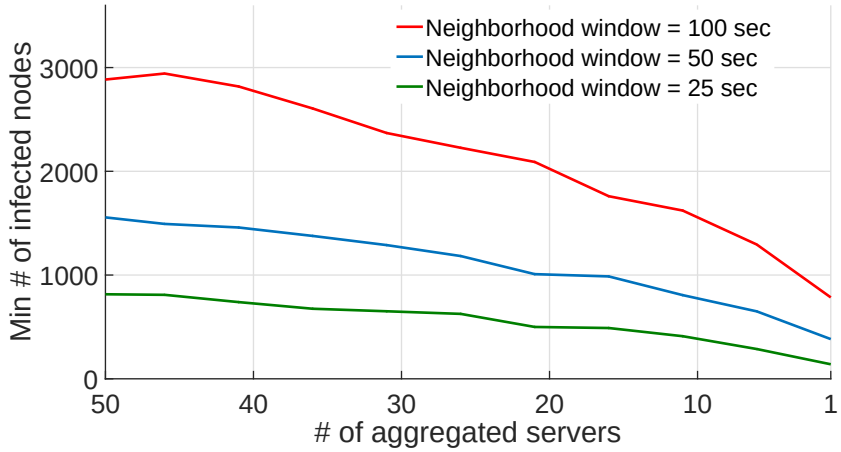

Figure 3.10: (Waterhole attack) Comparing to pure time-based NF, structural filtering algorithm improves Shape GD's performance by 3.75× – 5.8× by aggregating alerts on a server basis.

with 25 second NTWs, the number of infected nodes at detection time is 139.9 – higher than the 107 infected nodes for temporal filtering with a *6 second* NTW (Figure 3.8). Temporal and structural filtering thus present different trade-offs between detection time and work performed by GD – their relative performance is affected by the rate at which true and false positive FVs are generated.

3.5.4 Fragility of Count-GD

A Count GD algorithm counts the number of alerts over a neighborhood and compares to a threshold to detect malware. This threshold scales linearly in the size of the neighborhood – we now experimentally quantify the error Count GD can tolerate in phishing (Figure 3.12) and waterhole (Figure 3.11) settings. Note that the error in estimating neighborhood size can be double sided – underestimates (negative error) can make neighborhoods look like alert hotspots and lead to false

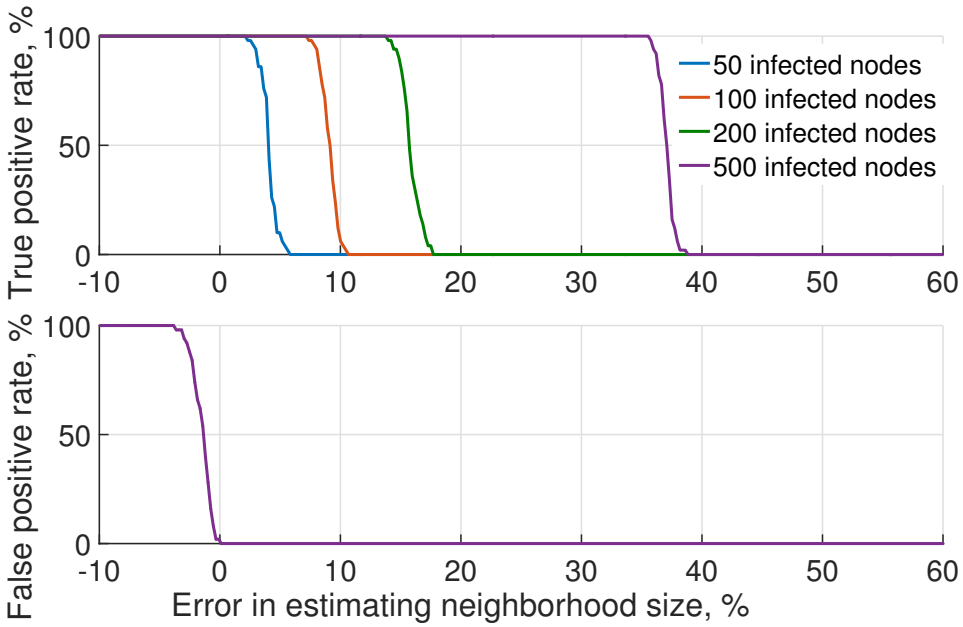

Figure 3.11: (Waterhole attack) An error in estimating neighborhood size dramatically affects Count GD's performance. It can tolerate at most 0.1% underestimation errors and 13.8% overestimation errors to achieve comparable with Shape GD performance.

positives, while overestimates (positive error) can lead to missed detections (i.e., lower true positives).

We run Count GD in the same setting as Shape GD when evaluating time-based NF (Section 3.5.2) – 30-sec long neighborhood including 17,178 nodes (Figure 3.11) to model a waterhole attack and 1-hour long neighborhood time window (NTW) with 1086 nodes (Figure 3.12 in Section F) to model phishing. We vary infection probability (waterhole) and click rate in emails (phishing) such that the number of infected nodes in a neighborhood changes from 0 to 55 (phishing) and from 0 to 500 (waterhole) in four increments – note that in both scenarios, only a small fraction (5.5% and 2.9%) of nodes per neighborhood get infected in the worst

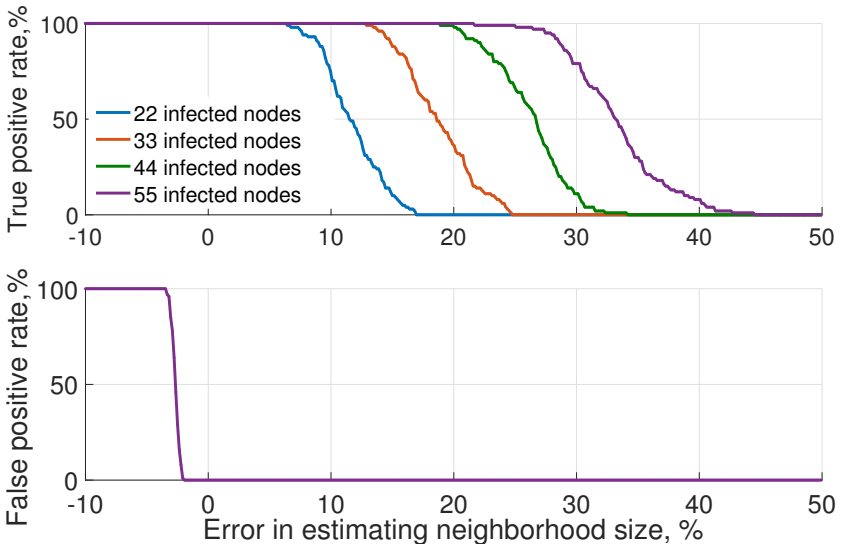

Figure 3.12: (Phishing attack) An error in estimating neighborhood size dramatically affects Count GD's performance. It can tolerate at most 2% underestimation errors and 6.3% overestimation errors to achieve comparable with Shape GD performance.

case.

In this setting, recall that the Shape GD has a maximum global false positive rate of 1% and a true positive rate of 100% – and detects malware when only 22 (phishing) and 200 (waterhole) nodes are infected – for the same NTWs. When the same number of nodes are infected, and for a similar detection performance, our experiments show that the Count GD can only tolerate neighborhood size estimation errors within a very narrow range – [-2%, 6.3%] (phishing) and [-0.1%, 13.8%] (waterhole). A key takeaway here is that underestimating a neighborhood's size makes Count GD extremely fragile (-2% in phishing and -0.1% for waterhole). On the other hand, overestimating neighborhood sizes decreases true positives, and this effect is catastrophic by the time the size estimates err by 17% (phishing) and 17.5%

(waterhole).

We comment that this effect can be important in practice. Given the practical deployments where nodes get infected out of band (e.g., outside the corporate network), go out of range (with mobile devices), or with dynamically defined neighborhoods based on actions that can be missed (e.g. neighborhood defined by nodes that 'open' an email instead of only downloading it from a mail server), the tight margins on errors can render Count GD extremely unreliable. Even with sophisticated size estimation algorithms, recall that the underlying distributions that create these neighborhoods (email open times, number of clients per server, etc) have sub-exponential heavy tails [108] – such distributions typically result in poor parameter estimates due to lack of higher moments, and thus, poorer statistical concentrations of estimates about the true value [81]. Circling back, we see that by eliminating this size dependence compared to Count GD, our Shape GD provides a robust inference algorithm.

3.6 How Accurate is Clustering for Global Malware Detection?

While Count GD is fragile, clustering GDs are inaccurate in the early stages of infection. This is why prior work [160] uses clustering to (offline) identify high-priority incidents from security logs for human analysis (instead of as an always-on GD) – this use case is complementary to an always-on global detector. We quantify a recent clustering GD's [160] detection rate on our data set.

First, we reduce dimensionality of 390-dimensional FVs by projecting them on the top 10 PCA components, which retain 95.72% of the data variance. Second,

we use an adaptation of the K-means clustering algorithm that does not require specifying the number of clusters in advance [100, 154, 160]. Specifically, the algorithm consists of the following three steps: (1) select a vector at random as the first centroid and assign all vectors to this cluster; (2) find a vector furthest away from its centroid (following Beehive [160], we use L1 distance) and make it a center of a new cluster, and reassign every vector to the cluster with the closest centroid; and (3) repeat step 2 until no vector is further away from its centroid than half of the average inter-cluster distance.

The evaluation settings of the clustering algorithm match exactly the settings where Shape GD detects infected neighborhoods with 99% confidence. Specifically, the algorithm clusters the data that we collected in a 17,178-node neighborhood under a waterhole attack within 30 seconds and the data that we collected over an hour-long session across 1086 nodes in a medium size corporate network under a phishing attack (Section 3.4.3). As we have already demonstrated (Section 3.5.2), Shape GD starts detecting malware when 107 (waterhole attack) and 22 (phishing attack) nodes get compromised (as in experiments for Figures 3.5 and 3.6).

Clustering does not fare well, and results look very similar for both waterhole and phishing experiments. The clustering algorithm partitions waterhole data set into 30 clusters. We observe three large clusters that aggregate most of the benign FVs. However, the algorithm fails to find small 'outlying' clusters consisting of predominantly malicious data. As for the phishing experiment, we observe a similar picture: the algorithm forms slightly higher number of clusters – 33 rather than 30 – and it identifies 4 densely populated clusters. In both cases each clus-

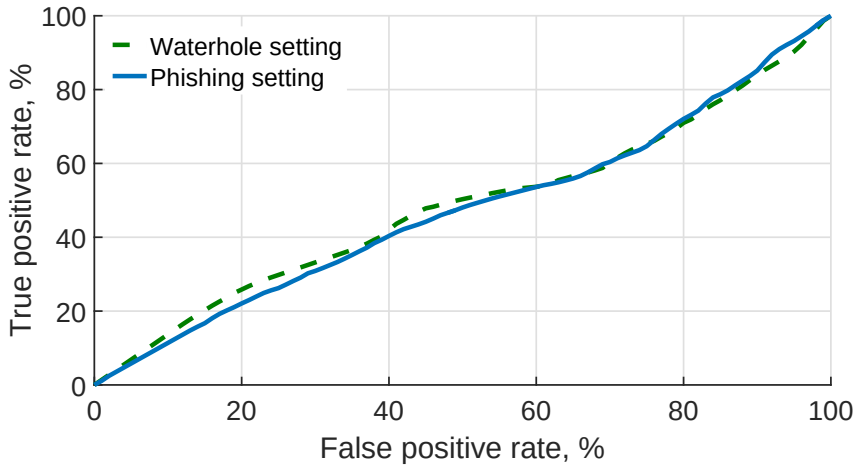

Figure 3.13: (ROC curve)bp True positive v. False positive curve shows detection accuracy of the clustering-based malware detector [160]. Its Area Under the Curve (AUC) parameter averaged for 10 runs reaches only 48.3% and 47.4% in the case of waterhole and phishing attacks respectively; such low AUC value makes it unusable as a global detector.

ter heavily mixes benign and malicious data, hence the clustering approach suffers from poor discriminative ability, i.e. it is unable to separate malicious and benign samples.

Note the clustering algorithm enforces explicit ordering across the clusters. That is, the algorithm forms a new cluster around an FV that is furthest away from its cluster centroid. Thus, earlier a cluster is created, the more suspicious it is. By design of the clustering algorithm, the clusters are subject to a deeper analysis in order of their suspiciousness. Such an inherent ordering allows us to build a receiver operating curve (Figure 3.13) and compute a typical metric for a binary classifier – Area Under the Curve (AUC) by averaging across 10 runs. The AUC reaches only 48.3% and 47.4% for waterhole and phishing experiments respectively.

This experiment illustrates the failure of the traditional recipe of dimensionality reduction plus clustering. There is a fundamental reason for this – the neighborhoods we seek to detect are small compared to the total number of nodes in the system. Optimization-based algorithms that exploit density, including K-Means and related algorithms, fail to detect small clusters in high dimensions, even under dimensionality reduction. The reason is that the dimensionality reduction is either explicitly random (e.g., as in Johnson-Lindenstrauss type approaches), or, if data-dependent (like PCA), it is effectively independent of small clusters, as these represent very little of the energy (the variance) of the overall data. Spectral clustering style algorithms [65, 119, 148] are also notoriously unable to deal with highly unbalanced sized clusters, and in particular, are unable to find small clusters.

Shape GD also reduces dimensionality but does so after neighborhood filtering. This amplifies the impact of small neighborhoods. The combination of dimensionality reduction, small-neighborhood-amplification, and then aggregation represents a novel approach to this detection problem, and our experiments validate this intuition.

3.7 How Many FVs does Shape-GD Need to Make Robust Predictions?

The number of FVs per neighborhood required by Shape-GD to make a robust prediction is a crucial parameter. With too few FVs produced by a neighborhood, benign neighborhoods' ShapeScore will have high variance (i.e. benign distribution in Figure 3.4 becomes wide and the gap between two distributions

shrinks), leading to global false positives and negatives. On the other hand, if neighborhoods are large, their ShapeScores will be dominated by the large number of benign FVs and thus lead to missed alerts (false negatives) especially in the early stages of infection. Further, the number of alert-FVs generated by a neighborhood in a deployed Shape-GD need to be comparable to or larger than those used in training – hence, we want to determine the smallest number of FVs Shape-GD needs to make a robust prediction.

Figure 3.14 shows the sensitivity of Shape-GD to neighborhood size (i.e., the number of FVs generated by nodes in a neighborhood during training stage). We vary the number of FVs that a neighborhood generates from 3,000 up to 30,000 FVs and average the results of 10 experiments. We present two metrics in Figure 3.14 – the red curve plots the inter-class distance (between histograms of benign and malicious neighborhoods from Figure 3.4), and the blue curve plots intra-class distance (i.e. the width of the benign histogram). Figure 3.14 shows that the red inter-class distance increases (and blue intra-class variance decreases) quickly as neighborhood size increases, and both curves flatten out once the neighborhoods start generating more than 15,000 FVs.

This shows that (for our Windows programs dataset) neighborhoods generating 15,000 FVs or more are a good choice to train Shape GD because purely malicious or benign distributions stabilize at this size. In real scenarios with mixtures of mostly benign and a few malicious neighborhoods, the number of FVs will have to be scaled up depending upon the timescale of attacks (hours for phishing v. seconds for waterhole) and the number of nodes affected by an attack (tens of

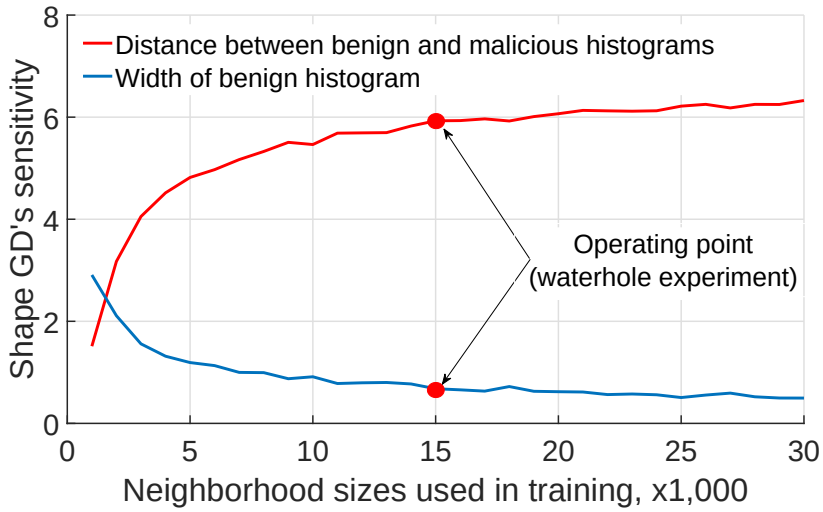

Figure 3.14: Analysis of ShapeScore histogram parameters when changing neighborhood size. The curves flatten out on the right side from the operating point.

nodes in enterprise email networks v. thousands in a broader waterhole attack on the enterprise). In the phishing and waterhole attack case studies in the paper, we use neighborhoods of 1,086 and ∼17,000 nodes that produce 15k FVs and 100k FVs respectively.

3.8 Computation and Communication Costs of Shape-GD

Local detectors. Generating a single FV, which is a 1-sec histogram of system calls, on a local host is equivalent to performing 2,500 (system call frequency) direct table lookups on average and incrementing corresponding counters. Projection on a PCA basis requires computing 10 dot products. Finally, running an LD, which is Random Forest in our case, results in performing 330 scalar comparisons on average. At 1 second per FV, the overheads of such an LD are negligible.

Data transfer. Each FV is composed of 10 floating point numbers (40 bytes total if assuming single precision format). In the phishing experiment 1086 hosts transfer (in aggregate) $\sim 40KB/sec$; data transfer rate in waterhole setting is a little bit higher: $\sim 4,450$ hosts transfer (in aggregate) $\sim 174KB/sec$. In both cases we assume Shape GD using pure time-based filtering with 1 hour and 6 sec neighborhood time windows respectively.

If Shape GD employs structural filtering on top of the time-based one, then data transfer depends on the number of emails floating in a network or on the number of servers. In both cases, data transfer scales linearly with the number of emails and servers. When applying the most fine-grained structural filtering in our experiments, the nodes susceptible to phishing attacks transfer $\sim 4KB/sec$ per email and the nodes susceptible to waterhole attacks send $\sim 40KB/sec$ per server when using 1 hour and 25 sec neighborhood windows respectively.

Server computations. After receiving a batch of alert-FVs, Shape GD performs lightweight computations. Overhead of binning scales linearly with the number of alert-FVs in a batch; each binning operation is a direct table lookup together with counter increment. Calculating ShapeScore, which is Wasserstein distance, results in a sequence of addition operations, whose total number is equal to the dimensionality of FVs, which is 10, multiplied by the number of bins, which is 50. To summarize, Shape GD's computational requirements are fairly light-weight.

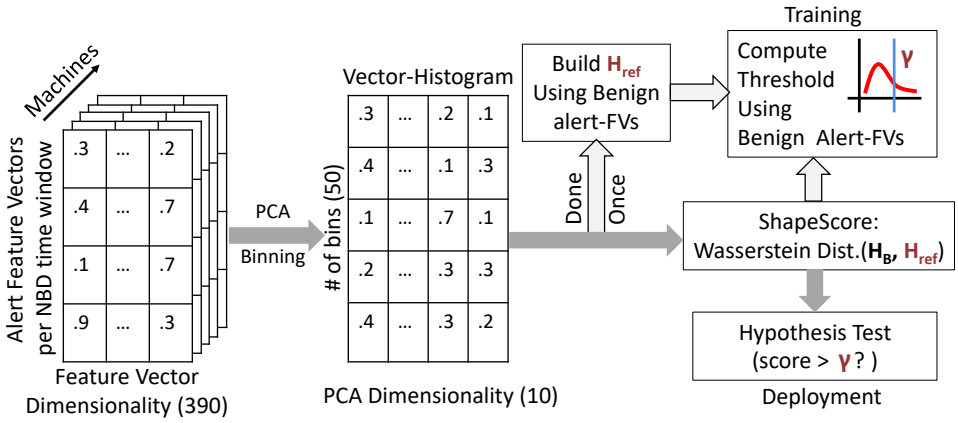

Figure 3.15: (Overview) Shape-GD machine learning pipeline.

3.9 Discussion

Global FPs vs LD FPs. As remarked in the Introduction, an FP of 1% at the global level means that we will see one alert every 100 - 300 hours (for the phishing scenario) and 100 seconds (for waterhole scenario the neighborhood time window slides by 1 second). This reduces work to be performed by the deeper, second-level analysis considerably.

Specifically, LDs operating at 6% false positive rate generate 23.5M – 70M FPs within 100–300 hours time interval in a network of 1086 nodes (phishing) and 300K alerts within every 100 sec interval where neighborhoods include ∼50K nodes on average (waterhole). Shape GD filters these alerts. When using 1–3 hours (phishing) and 6 sec (waterhole) time-based neighborhood filtering, Shape GD will report to a system running a deeper analysis approximately 234.5K – 703.7K FPs raised by LDs (phishing) and approximately 1.4K FPs (waterhole). Adding structural filtering brings these numbers down to 21.6K – 64.8K FPs (phishing) and 360 FPs (waterhole).

Compared to a neighborhood of LDs, Shape GD thus reduces the number of FPs reported to deeper analyses by $\sim$100$\times$ and $\sim$200$\times$ when employing time-based filtering only (for phishing and waterhole scenarios respectively), while structural filtering reduces alert-FVs for deeper analysis to $\sim$1000$\times$ and $\sim$830$\times$. In both scenarios, analysts can choose to reduce number of alert-FVs to be analyzed by sliding neighborhood windows by a larger interval; however, this will increase the time to detect malware infection.

Shape property across LDs and platforms. Shape-GD relies on conditional separability of FPs and TPs, and we use only one LD type for evaluation – a system call histogram-based LD. However, we have experimentally determined that FPs and TPs are separable for other LD types as well – an n-gram-based LD [95] and an LD that uses VirusTotal [46] reports for malware detection [117]. Further, we can classify malicious neighborhoods on the Android platform – using malware binaries obtained from the NCSU dataset and contagio dump website, and using benign applications that we drive using real human user input – in addition to the Windows setup that we describe here. We have left out the details due to lack of space but can produce an anonymous report if requested.

Though the local detectors we built have a 6% FP rate, Shape-GD can work well work with better LDs. Shape-GD only requires LD's FPs and TPs to be separable and to be able to aggregate enough alert-FVs across the nodes within a neighborhood. We deliberately do not consider rule-based LDs that are commonly used within enterprise networks because, even though their FP rate is very low, they suffer from a high false negative rate, and they can be easily evaded with simple

malware transformations.

Performance Overheads. Recall that Shape GD requires only alert FVs – this leads to a two-fold dimensionality reduction when sending data from individual LDs to the GD. First, the FVs are low-dimensional (here, 10-dimensional vectors). Second, only alert FVs are needed – this leads to a 16-fold reduction in data (roughly only 6% of the FVs lead to alerts). Further, the Shape GD is a batch processing algorithm, thus, the individual nodes can batch their data at coarse timescales (e.g. once every NTW) and send the data to the Shape GD. Finally, it does not matter even if some batches are lost/missed; recall that the Shape GD is robust to precisely this type of noise. Appendix A discusses overheads in more depth but the key takeaway is that Shape GD has low overheads – each LD can use simple dot products and scalar comparisons to implement PCA and Random Forests, the total incoming bandwidth to the Shape GD server ranges from 40KBps to 174KBps for phishing and waterhole respectively, and the server only needs to bin data (into 50 bins) and compute Wasserstein distance (add 10 counters in each bin).

Detailed Shape-GD Pipeline. As an extension to the description in Section 4, Figure 3.15 shows the detailed machine learning pipeline for extracting one neighborhood's shape into a ShapeScore.

3.10 Conclusions

Building robust behavioral detectors is a long-standing problem. We observe that attacks on enterprise networks induce a low-dimensional structure on

otherwise high-dimensional feature vectors, but this structure is hard to exploit because the correlations are hard to predict. By analyzing alert feature vectors instead of alerts and filtering the alert-FVs along neighborhood lines, we amplify the signal buried in correlated feature vectors, and then use the notion of statistical shape to classify neighborhoods without having to estimate the expected number of benign and false positive FVs per neighborhood. We note that both neighborhood-filtering and shape are complementary techniques that apply across a range of LDs or platforms – e.g., we have determined that Shape-GD also works well with n-grams based LDs (instead of histograms) and on the Android platform (in addition to Windows).

Our methodology composes the traditional host-level malware analysis methodology with trace-based simulations from real web services (to overcome the lack of joint LD-GD datasets), and allow us to run sensitivity analyses that will be precluded by using an actual enterprise trace. We find that Shape-GD reduces the number of FPs reported to deeper analyses by $\sim$100$\times$ and $\sim$200$\times$ when employing time-based filtering only (for phishing and waterhole scenarios respectively), while structural filtering reduces alert-FVs to $\sim$1000$\times$ and $\sim$830$\times$ (Appendix 3.9). Neighborhoods and their shape thus serve as a new and effective lens for dimensionality reduction and significantly improve false positive rates of state-of-the-art behavioral analyses. For example, LDs can operate at a higher false positive rate in order to reduce false negatives and improve computation efficiency.

Chapter 4

The Shape of Alerts: Detecting Malware Using Distributed Detectors by Robustly Amplifying Transient Correlations in the Symantec Wine Dataset

4.1 Introduction

This chapter describes a malware-detection framework, Centurion, that takes ideas outlined in the Shape-GD project (Chapter 3) to the next level and performs real-time malware detection in the Symantec Wine dataset. Thus, Centurion operates in noisy communities of weak behavioral detectors with long-term log entries from the Symantec antivirus detector. For completeness, here we only summarize Shape-GD's concepts – neighborhood filtering and shape properties. A detailed discussion is in Chapter 3.

Shape-GD's ideas: Neighborhood filtering and shape. We hypothesize that weak local detectors can be aggregated robustly by using information about how malware spreads. Our proposed system, Centurion, relies on two key insights to correctly identify malicious feature vectors.

First, although attacks can take many forms, attack vectors are easier to identify. For example, many attacks on Symantec's client machines rely on 'downloader trojans' to bring successive stages of payloads – hence, *downloader graphs* [110]

on a machine are correlated with malware propagation. Similarly, in a firewalled enterprise, machines that visit a specific server (in watering hole attacks) are more likely to be compromised than a random machine in the enterprise.

Our key assumption is that *machines that have been exposed to a common attack vector have correlated alerts*. We call such a set of machines a *neighborhood*. Grouping local detectors into neighborhoods (as they form dynamically) concentrates a signal of malware activity that is otherwise not visible at the overall community level. However, neighborhoods are extremely noisy due to exploit types, machine status, and human usage and render cluster- and count-based GDs ineffective; hence we propose Centurion to aggregate local detectors' (LDs') output.

The second insight behind Centurion is that the *distributional shape of a set of suspicious feature vectors can robustly separate true-positive neighborhoods from false-positive neighborhoods*. Centurion analyzes only those feature vectors that cause alerts from LDs (*alert FVs*) rather than analyzing all feature vectors. Alert FVs thus represent draws from one of two *conditional distributions* – i.e., distributions of malicious or benign feature vectors that are conditioned on being labeled as malicious – that are similar but are not the same. Next, while a single suspicious feature vector is uninformative, a set of such feature vectors (i.e., alert FVs from a neighborhood) can indeed be tested to come from one of two similar-but-distinct distributions.

Overview: Malware detection in the Symantec Wine dataset. We evaluate Centurion on 5 million client machines monitored by malware detectors (in this case,

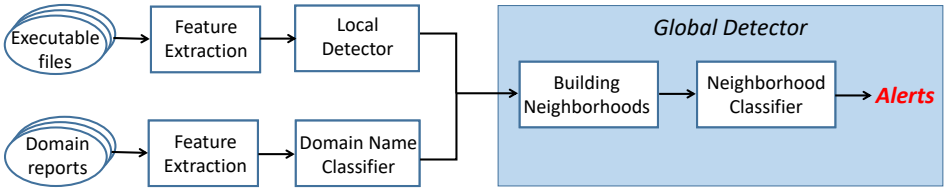

Figure 4.1: Application of Centurion to malware detection in the Symantec Wine dataset.

Symantec [50]). An LD algorithm [117] applied to the Symantec Wine dataset [50] analyzes file attributes using VirusTotal and achieves a false-positive rate of 5%. With 5 million local detectors in place, this requires a deeper human or program analysis – of up to ∼1.1M files to detect close to 137K malware files. A recent local detector reduces false-positive rates to 1% by training on metadata, such as features extracted from downloader graphs [110, 111]. However, this increases false negatives because it only detects malicious downloaders (those that install malware on devices), which comprise only ∼32.7% of the overall malware in the community.

4.2 Centurion: Algorithm

Though Centurion shares a similar design with Shape-GD, its internal implementation is significantly different. The algorithm consists of feature extraction, LDs, and a global detector (GD). Figure 4.1 visualizes Centurion's implementation. Our key innovations are in the GD. The LD design is inspired by prior work [117]; we briefly summarize LD detection performance in Section 4.4.

4.2.1 Centurion: Classifiers

Centurion employs a local detector that analyzes executable files and a domain name classifier that analyzes domain metadata to establish domain reputation.

File-level local detectors. LDs primarily rely on a lightweight static analysis, which scales well for processing millions of downloads per day. They are designed to run existing commercial tools such as TRID, ClamAV, and Symantec on a binary file; analyze statically imported libraries and functions; detect common packers; check whether a file is digitally signed or not; and collect the file' binary metadata. However, the Symantec Wine dataset lacks executable files (it includes only their hashes). VirusTotal, on the other hand, provides outputs of these tools as a single file-level report, which we directly use as LD input.

The core LD component is a feature-extraction algorithm, i.e., an algorithm that converts textual VirusTotal reports into fixed-length feature vectors, which are used as input by a classifier. Our LDs combine feature extraction described in a prior work [117] and a standard machine-learning classifier, XGBoost [64]. Even though the original algorithm [117] allows the classifier to achieve high accuracy, it is inappropriate in our case because it produces very high-dimensional feature vectors that have limited usability due to the 'curse of dimensionality' and high memory consumption. Instead, we use a common method for dimensionality reduction – feature hashing [151]. Empirically we found that the output dimensionality of 1024 allows us to achieve the best trade-off between LDs' accuracy and resource consumption.

Domain name classifier. To start using Centurion, a human analyst needs to supply a description of neighborhood attributes. This can be as simple as a list of high-valued servers (watering hole attacks), or it can be derived using a machine-learning algorithm. We use a domain name classifier, which uses VirusTotal domain reports as input to detect suspicious domains that are used to form neighborhoods.

The domain name classifier analyzes VirusTotal domain reports and identifies domains that are likely to distribute malicious files. It also aggregates domain classification produced by other commercial tools such as Dr. Web, Websense ThreatSeeker, and VirusTotal (Table 4.2). Each of these tools categorizes a domain based on its content. The number of categories ranges from 55 (Dr. Web) to 451 (VirusTotal), and they include classes such as social networks, banking, ads, government, etc.

The domain-name classifier applies one-hot encoding schema to represent categorical data as fixed-length feature vectors. Specifically, it creates a 'zero' feature vector with the number of elements equal to the total number of categories (767-dimensional feature vectors, in our case) and sets 'one' in the positions corresponding to the assigned categories – which are not necessarily mutually exclusive. The classifier (XGBoost [64]) uses numeric fields as they are (without additional encoding). The classifier also considers VirusTotal domain-related statistics such as the number of malware samples distributed by a particular domain, the number of samples that refer to a domain, and the number of malicious URLs belonging to a domain. These numeric values lie in the range from 0 to 100. (They seem to be capped at the 100 level.) The statistic is aggregated across the entire observa-

Categorical Domain Attributes		
Tool name	**# of categories**	**Examples**
Dr. Web	55	Social networks; weapons; violence; e-mail; chats
Websense ThreatSeeker	253	Dynamic dns; government; military; advocacy groups
VT categories	451	Ads; bank; outsourcing; webmail; web analytics
Numeric Domain Attributes		
Category name	Range	
Detected samples	[0,100]	
Referrer samples	[0,100]	
Detected URLs	[0,100]	

Figure 4.2: Domain name classifier's features.

tion period, i.e. since the first time a domain name was submitted to the VirusTotal service.

4.2.2 Neighborhood Instances from Attack-Templates

Within each neighborhood time window (NTW), Centurion generates neighborhood instances based on statically defined attack vectors; each attack vector is a template for generating neighborhoods. The goal of partitioning data into neighborhoods is to create predominantly benign or malicious neighborhoods. The algorithm runs once per NTW.

Algorithm 4 partitions downloaded files into multiple neighborhoods. It uses the following logic: if a domain is malicious, then the files transitively downloaded from such a domain are also likely to be malicious.

For ease of explanation, we treat the previously introduced domain name classifier as a predicate (line 1). At each iteration, the algorithm starts by identifying

a set of suspicious domains within the current NTW (lines 4–5), which is done using the domain name classifier. Then the algorithm uses each suspicious domain as a seed to initiate the neighborhood-formation process (lines 6–12). Next, for each suspicious domain, the algorithm searches for files within the current NTW that access that particular domain (files that are either downloaded from it or are being downloaded from it), resulting in set F (line 7). By following downloader graph edges, the algorithm selects files transitively downloaded by the files in set F (line 10) and filters out those that do not access any of the suspicious domains (line 11). The files that have not been excluded are added to the current neighborhood (line 12).

Note that the algorithm's formation process may generate many small neighborhoods. An estimate of the conditional distribution using such feature vectors (Section 4.2.3) is usually susceptible to high variance; thus, neighborhoods containing an insufficient number of files may have a negative impact on the accuracy of the neighborhood classifier (Section 4.2.3). To reduce variance and achieve robust classification of neighborhoods, the algorithm merges them such that final neighborhoods are greater than a predefined minimum size. Empirical analysis of the neighborhood classifier's accuracy shows that it achieves robust classification of neighborhoods containing more than 1000 files.

In order to maintain the neighborhood effect after merging – i.e., to have mostly homogeneous neighborhoods, either benign or malicious – the merging algorithm ranks neighborhoods in terms of maliciousness, where the malicious score is defined as the relative number of LD alerts within a neighborhood. After that, the

Algorithm 4: Symantec Wine: Neighborhoods from Attack-Vectors

Input : Downloader graphs
Output: Neighborhoods

Domain name classifier
Let DNC (*domain*): *domain* is malicious

execute once per NTW
while *True* **do**
    create an empty list of neighborhoods
    $nbds \leftarrow \emptyset$
    identify active domains within the current NTW
    $D \leftarrow$ domains accessed within the current NTW
    identify suspicious domains
    $D' \leftarrow \{\text{d} \in D \mid \text{DNC}(d)\}$
    foreach *suspicious domain* $d_i \in D'$ **do**
        identify files accessing the domain d_i
        $F \leftarrow$ files accessing the domain d_i
        initialize an empty neighborhood
        $nbd \leftarrow \emptyset$
        foreach *file* $f_i \in F$ **do**
            search for transitively downloaded files
            $F_i \leftarrow$ files transitively downloaded by f_i
            retain suspicious files
            $F_i' \leftarrow \{file \in F_i \mid \exists d \in file.domains \quad DNC(d)\}$
            $nbd \leftarrow N \bigcup F_i'$
        $nbds \leftarrow nbds \bigcup nbd$

algorithm sorts neighborhoods based on their malicious scores and merges them in decreasing order of their malicious scores. Note that the malicious score estimation may be incorrect if we incorrectly estimate the neighborhood size. However, Centurion tolerates such errors.

4.2.3 Shape Property for Malware Detection

After the neighborhoods are indentified, the next step is to detect neighborhoods with a high malware concentration. In order to accomplish this, we introduce *a novel approach to extracting neighborhood features* that formalizes the *shape property*.

The *key algorithmic idea* is to map all alert FVs within a neighborhood to a *single vector histogram* that robustly captures the neighborhood's statistical properties. *Such transformation allows us to analyze the joint properties of all alert FVs generated within a neighborhood without requiring FVs to be clustered or alerts to be counted.* After that, Centurion feeds neighborhood-level feature vectors into a binary classifier to identify malicious neighborhoods. We use a boosted decision tree classifier (XGBoost [64]) in our experiments.

Generating a vector-histogram from alert-FVs. The algorithm aggregates L-dimensional projections of alert FVs on a per-neighborhood basis into a set B (Algorithm 5, line 3). After that, Centurion converts a low-dimensional representation of alert FVs, the set B, into a single (L,b)-dimensional vector histogram, denoted by H_B (line 4). The conversion is performed by binning and normalizing L-dimensional vectors within the set B along each dimension. Effectively, a vector histogram is a matrix $L \times b$, where L is the dimensionality of alert FVs and b is the number of bins per dimension. Further implementation details can be found in Section 4.3.2.

We use standard methods to determine the size and number of bins. In

particular, we tried square-root choice, the Rice rule, and Doane's formula [8] to estimate the number of bins. We found that 20–100 bins yield the best results.

Neighborhood classifier. Centurion may use any binary classifier (Algorithm 5, line 4) as a neighborhood classifier. For our purposes, we use boosted decision trees – specifically, the XGBoost algorithm [64]. The main advantage of using XGBoost over Shape-GD's ShapeScore function is its ability to learn complex decision boundaries. It can also be trained in a non-parametric mode. (We completely automated parameter search process.) However, in comparison to Shape-GD's ShapeScore, the XGBoost algorithm requires both benign and malicious data for training purposes. In our experiments, we found that XGBoost outperforms the ShapeScore function in the Symantec Wine experiment, whereas the ShapeScore yields good detection accuracy in watering hole and phishing experiments (Chapter 3).

Note that like any other machine-learning classifier, the binary classifier employed by Centurion needs to be retrained periodically to account for evolving statistical software properties.

4.3 Experimental Setup

We evaluate Centurion using a proprietary Wine dataset [50], released by Symantec for research purposes. Centurion uses malware reports from Symantec client devices and reduces LD false positives from $\sim$1M to $\sim$110K, while retaining 107K out of 137K malware files. Centurion successfully amplifies the weak signal inherent to malware propagation.

Algorithm 5: Neighborhood Classification

Input : Suspicious neighborhoods *nbds*
Output: Malicious neighborhoods

```
for each nbd in nbds do
    aggregate L-dim projections of alert-FVs on per neighborhood basis
    B ← {alert − FVs | alert − FV ⊂ nbd}
    build an (L,b)-dim. vector-histogram
    H_B ← bin & normalize B along each dimension
    classify the neighborhood
    if Neighborhood Classifier(nbd) then
        label nbd as malicious
```

4.3.1 Wine Dataset

The Wine dataset [74, 110, 111] contains telemetry information collected by Symantec's intrusion prevention system and antivirus products over a five-year period (from 2008 until 2013). The dataset summarizes file downloader a ctivities across 5M Windows hosts around the world. File downloads are represented in the form of downloader graphs (the abstraction introduced by Kwon et al. [110]) – one per end host. A graph node represents a downloaded file (SHA256 file hash), and a directed edge between two nodes n_a and n_b indicates that file n_a has downloaded file n_b from a domain D on the corresponding host machine, where D is the edge's label.

Figure 4.3 depicts an example of a downloader graph. Each node is labeled with a corresponding file name, and each edge bears a domain name from where the file has been downloaded. We also overlay ground truth on the nodes and edges: red indicates that a file or domain is malicious, while blue indicates that a file or domain is benign.

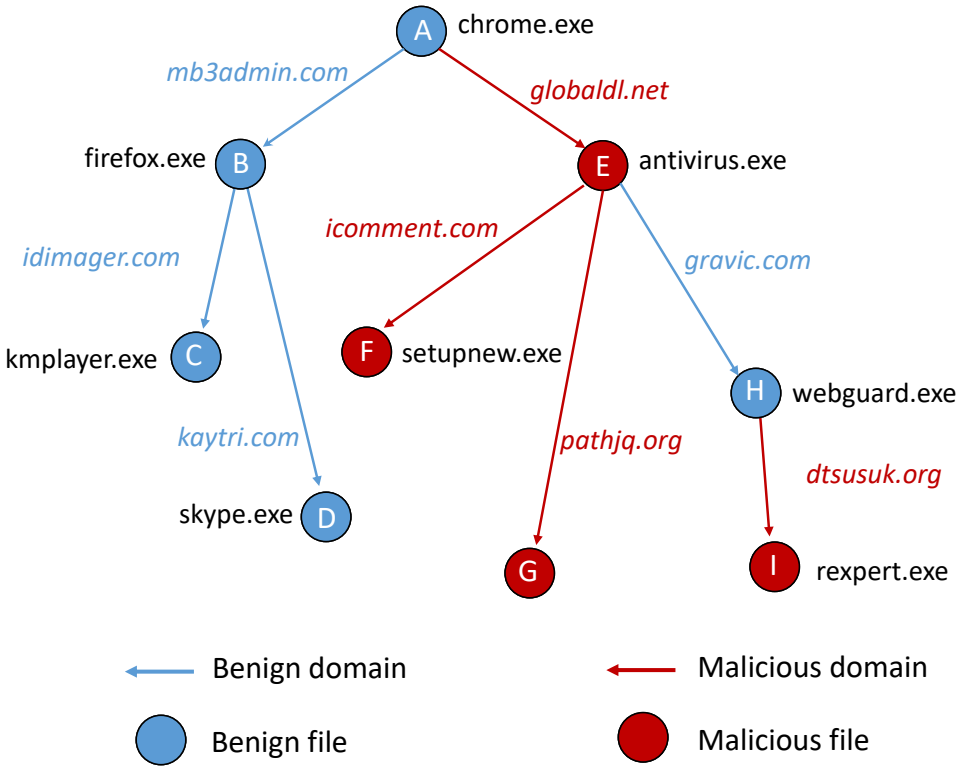

Figure 4.3: Example of a downloader graph.

We used the VirusTotal service to obtain ground-truth information about the 20.3M file hashes downloaded 67M times, as well as all 353K domain names in Wine (Table 4.7). Though file-level VirusTotal reports contain results of signature-based malware detection, we do not use them within Centurion (except for in computing ground truth). Hence, information within VirusTotal domain reports might be affected by post-analysis performed by commercial antivirus vendors. However, there exist alternative approaches to establishing domain reputation [92] that outperform our domain name classifier by using a different set of domain features that are unavailable in the Symantec Wine dataset.

For files (corresponding to a file hash) or domain names that VirusTotal has information for, it used 62 different antiviruses and other heuristics to generate a report. This report is used to train the file-behavior and domain-name classifiers. We consider a file to be malicious if more than 30% of antivirus products label it as malware [110]. This yields 2.6M reports for file hashes, with 137K confirmed to be malicious, and 301K reports for domain names. We label all remaining files and domain names (i.e., those that are not confirmed to be either malware or benignware by VirusTotal) as benign. This is a conservative step that weakens the malware propagation signal in the dataset and is also representative of real deployments where information about suspicious file or domain names is often delayed or unavailable.

Neighborhood example. After introducing a visual downloader graph abstraction (Figure 4.3), we describe an example showing how Algorithm 4, which builds neighborhoods, actually works.

For each domain name, the domain classifier outputs the probability of a domain being malicious. If a domain is considered malicious (e.g. globaldl.net), the algorithm starts forming a neighborhood by identifying all files that access that domain within an NTW (files A and E in our example), and for each of them the algorithm performs a breadth-first search (BFS) on the downloader graph. While traversing a downloader graph in a BFS manner, the algorithm uses domain name predictions produced by the domain name classifier to discard files downloaded from likely benign domains (e.g. gravic.com). Had the domain name classifier correctly identified all suspicious domains, then the neighborhood would include malicious files E, F, G, and I. (A is excluded because all browsers with reputable digital signatures are whitelisted.) However, due to the high false-positive rate of the domain name classifier, only 30% of the files in the neighborhood are malicious; the rest of the files are benign. According to Algorithm 4, a file is excluded from a neighborhood only if all its edges (incoming and outgoing) in the downloader graph are predicted to be benign. In the example in Figure 4.3, the neihgborhood formed by Algorithm 4 includes files E, F, G, H, and I. Note that file H is a false positive, but it is included because it accesses a suspicious domain, dtsusuk.org.

4.3.2 Vector-histogram Implementation

We separate implementation details of the algorithm that generates a vector histogram (Algorithm 4.2.3) from its design. Centurion deals with two types of alert FVs: file and domain. Therefore, it builds two separate vector histograms per neighborhood and then concatenates them into a single vector histogram. The file-

level vector histogram has a dimensionality of 10×50; i.e., each file alert FV is projected on a 10-dimensional basis and binned into 50 bins along each dimension. Similarly, a domain vector histogram has a dimensionality of 100×5; i.e., each domain alert FV is projected on a 100-dimensional basis and binned into 5 bins along each dimension. In each case, the basis consists of 10 and 100 principal components, respectively, computed using principal component analysis. We select the number of basis vectors such that they retain more than 95% of variance.

After building two separate vector histograms, the algorithm concatenates them into a single vector histogram. For this purpose, it represents them as two 500-dimensional vectors by using a row-major order and appends the second one to the first one. Thus, the resulting vector has 1,000 dimensions, which are used as input for the neighborhood classifier.

4.4 Results

We now quantify how Centurion concentrates malware in the Symantec Wine dataset into neighborhoods. By using downloader graphs as a weakly correlated attribute, Centurion identifies malicious files and infected machines with significantly lower false positives than LDs [117] alone and far higher true positives than a downloader-graph based-detector [110, 111] alone.

In addition, neighborhoods and shape together are good predictors of malware behavior – hence, Centurion does not have to wait until the entire sequence of malware payloads have been downloaded to declare a downloader or machine as malicious. We find that, on average, Centurion can identify a file as malicious only

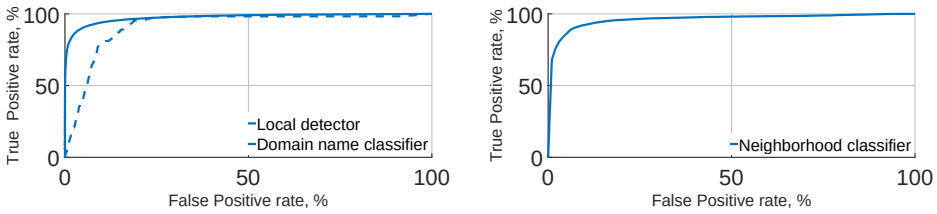

Figure 4.4: (Left) Receiver operating curve (ROC) of the local detector and the domain name classifier. (Right) ROC of the neighborhood classifier.

$\sim$20 days after it enters the Wine dataset and $\sim$345 days before VirusTotal confirms it as malware. Table 4.7 summarizes these results.

4.4.1 Centurion: Classifiers

Local detectors. We start with the evaluation of local detectors (Section 4.2.1). Each LD algorithm comprises two parts: feature extraction and a binary classifier (XGBoost in our prototype). We train an LD on a set of 2.6 million VirusTotal reports using 10-fold cross-validation. The detector achieves a 97.61% AUC metric (Figure 4.4), and we chose the operating points of a 5.0% false-positive rate and a 90.47% true-positive rate. Note that due to the high number of benign files in the dataset, a 5.0% false positive rate corresponds to more than 1M misclassified files, which is likely to prevent practical deployment of such a local detector. In subsequent experiments, we use out-of-fold predictions made by the detector.

Domain name classifier. We train and evaluate the classifier (Section 4.2.1) on 251K VirusTotal domain reports using 10-fold cross-validation to achieve a 91.58% AUC (Figure 4.4). We specifically choose an operating point of 19.03% false positives and 95.41% true positives.

The domain name classifier is weak because it is conservative in labeling domains – an entire domain is considered malicious if it serves at least one malware sample. However, even malicious domains serve several benign files, and the local detector (above) that analyzes file-level features using VirusTotal contradicts the domain name classifier. Adding more information about the URL can improve the classifier. However, even the weak signal in domain names is sufficient for Centurion to significantly improve the local detectors. Interestingly, because the domain name classifier is only used to create neighborhoods (and not alerts), it can operate at a conservative setting and rely on the shape-based neighborhood classifier to weed out false positives.

The domain name classifier lets Centurion efficiently filter out domains that are unlikely to distribute malicious files. Specifically, it removes from further consideration 68.62% (214,884 out of 313,133) completely benign domains that are responsible for delivering 80.70% (16,222,941 out of 20,103,211) benign files. At the same time, the classifier retains 75.86% (30,448 out of 40,134) malicious domains responsible for delivering 88.31% (94,457 out of 106,959) malicious files.

Neighborhood classifier. The neighborhood classifier (Algorithm 5) performs neighborhood-level feature extraction and feeds resulting feature vectors into an XGBoost classifier. We estimate its detection capabilities using 10-fold cross-validation. The ROC plot (Figure 4.4) shows that the classifier achieves a 96.13% AUC score, and we choose the following operating points: 5% false positives and 91.83% true positives.

A neighborhood-level alert is different from the above file- and domain-

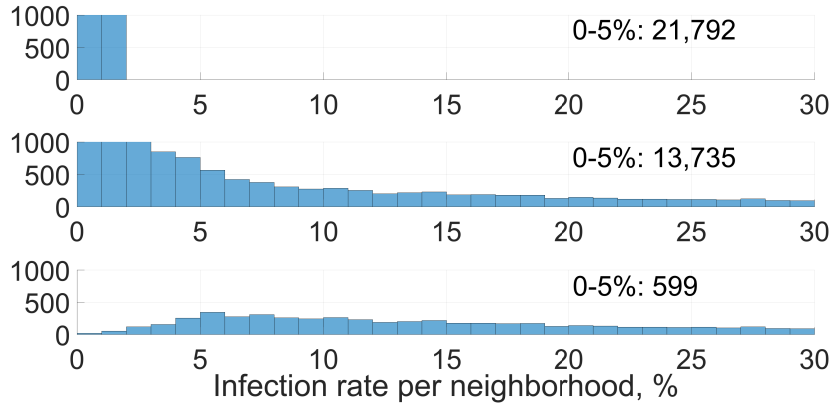

Figure 4.5: Neighborhood classifier acts as a malware concentrator. (Upper) Distribution of infection rates of randomly grouped files. (Middle) Distribution of neighborhoods' infection rates. (Lower) Distribution of neighborhoods' infection rates after filtering out low-infected neighborhoods. The neighborhood classifier retains only highly infected neighborhoods. (Distributions are capped at 1,000 level.)

name-based LD alerts. It signifies that a set of files that have suspicious behavior have been downloaded from suspicious links, and hence identifies the large majority of files that were false positives at the local level. First, we measure the degree to which our neighborhood classifier removes benign files, and then show that by re-examining files in suspicious neighborhoods (using the file-based LD), we can capture 78.03% of true positives.

4.4.2 Neighborhoods Concentrate Malware

First, we measure the effect of using domain names from downloader graphs as an attribute to create neighborhoods.

The original malware concentration in the Wine dataset is only 0.663%, as shown in the top-most plot of Figure 4.5. If a random subset of files are grouped

into a neighborhood, each neighborhood will have considerably less malware than the false-positive rate of the malware detectors (5%) – i.e., creating neighborhoods randomly does not concentrate malicious activity. This is the baseline against which downloader-graph-based neighborhood creation and shape-based neighborhood classifiers must be compared; the neighborhoods labeled as malicious must contain more than 5% malicious files while achieving high malware coverage overall.

Centurion first uses the domain name classifier to prune out files downloaded from benign domains; this increases the 0.663% infection rate to 9.49% (middle plot, Figure 4.5).

Centurion then uses the shape-based neighborhood classifier to identify infected neighborhoods. This dramatically changes the distribution of neighborhood infection rates; i.e., the peak shifts to the right, from 1% to 5% (lower plot in Figure 4.5). The neighborhood classifier brings the average malware concentration in a neighborhood from 9.49% to 24.6% – an increase of 37.1$\times$ compared to randomly grouping files into neighborhoods.

Specifically, the number of neighborhoods with an infection rate of less than 1% drops by 437.6$\times$ (from 8752 on the upper plot to 20 on the lower plot). Overall, the neighborhood classifier together with the domain-name classifier reduce the number of low infected neighborhoods (neighborhoods with less than 5% of malicious files) by 36.4 times (from 21,792 to 599).

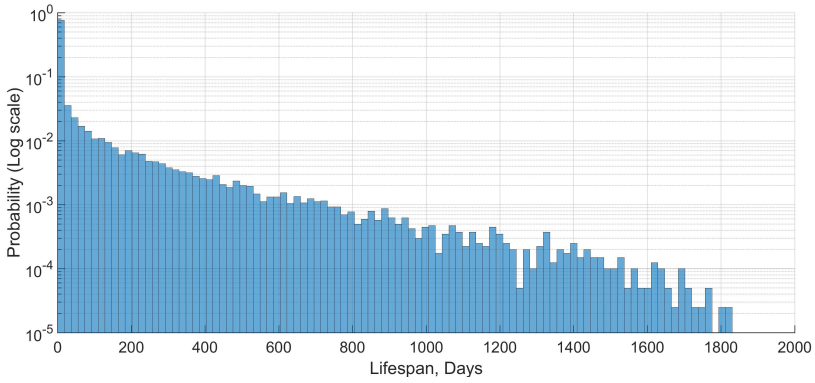

Figure 4.6: Distribution of the lifespan of malicious domains. After removing domains serving only a single malicious file, the average lifespan goes up to 157 days. And we set NTW to 150 days.

4.4.3 Lifespan of Malicious Domains.

Before discussing detection results, we analyze the lifespan of compromised domains to justify our choice of the NTW (150 days). We define the lifespan of a malicious domain as a time interval between the first malware sample and the last malware sample distributed by the domain. In practice, a domain may be compromised multiple times within its lifespan. Unfortunately, the Wine dataset lacks information necessary to distinguish between separate compromises. However, such a coarse-grained definition of the lifespan parameter allows us to conservatively estimate how long a domain remains compromised or how long a malicious domain remains active if its sole purpose is to distribute malware.

Figure 4.6 shows the distribution of domain lifespan on a log scale. We split the range of the lifespan parameter (from 0 days up to 1822 days) into 100 bins (horizontal axis). For each bin we compute the probability that a lifespan value

will fall into it (vertical axis). We use the log scale to better visualize the long-tale distribution of lifespan values.

A large portion of malicious domains, specifically, 62.4%, in the Symantec Wine dataset serves only a single malicious file. We are likely to observe such an effect due to the dataset being incomplete, i.e., it does not capture all malicious downloads, and hackers are also actively employing domain-generation algorithms [129] that register domains serving few malicious files. Across the dataset, the average lifespan of malicious domains is 59 days and the median is 0 days. If we remove domains serving only a single malicious file, the average lifespan goes up to 157 days and the median is 44 days.

Therefore, we consider file downloads within a 150-day time window to be correlated and set the NTW in our experiments to 150 days. However, if we had more detailed information on individual domain compromises in the dataset, we would be able more accurately estimate the average lifespan and, thus, the average size of an NTW within which malicious downloads are actually correlated. More accurate estimations of the NTW length would improve overall malware detection.

4.4.4 Aggregate Detection Results

We now quantify the detection performance of the complete pipeline by applying the malware classifier to files inside infected neighborhoods. By identifying malicious neighborhoods, Centurion effectively weeds out many files that trigger false alerts; hence, the alerts within infected neighborhoods are $\sim$37 times more likely to be malware (true postive).

To perform real-time analysis, we replay the five-year history of download events in the Wine dataset (each event has a time stamp associated with it) and execute Centurion every 30 days. We set the NTW parameter to 150 days because we found that the average lifespan of malicious domains is 157 days. In our experiments, we observed that a shorter time period between consecutive runs of Centurion does not significantly affect results, it only improves time to detection and early detection parameters (Table 4.7). We intentionally stick to a 30-day period between consecutive runs of Centurion to keep execution time ($\sim$12 hours) and resource consumption manageable.

We compare Centurion, which comprises the neighborhood classifier and LDs, with prior work – LDs [117] and the state-of-the-art malware detector in the Wine dataset [110] – as well as with a neighborhood detector. For comparison, we use standard machine-learning metrics: $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, and $F-1 = 2 \cdot \frac{precision \cdot recall}{precision+recall}$.

All four detectors explore different operating points in the true-positive/false-positive (FP/TP) design space. To compare them, we use a standard machine-learning metric: the F-1 score, which is a harmonic mean of precision and recall. The F-1 score is bounded by 100%, which is achieved only if a detector has a 100% TP rate and 0% FP rate.

4.4.5 File-level Aggregate Results

Though Centurion is designed to act as a real-time malware detector, i.e., output detection results every time it is executed, in this section we only focus on

Parameter	Prior Work [CCS'15]	Local Detectors [DIMVA'16]	Neighborhood Detector	Centurion
# of benign files	20,407,667			
# of malicious files	137,279			
Possible early detection, files	68,398			
False positives	4,390	1,022,439	1,184,234	109,951
	0.021%	5.01%	5.80%	0.54%
True positives (Recall)	43,091	124,190	115,357	107,124
	31.39%	90.47%	84.03%	78.03%
Precision	73.5%	10.83%	8.88%	49.35%
F-1 score	46.64%	19.35%	16.06%	60.46%
Early detection, days	9.24	214.08	340.42	345.33
Time to detection, days	N/A	0	20.46	20.33
Early detection, files	3,002	62,815	31,266	29,356
	4.4%	92%	46%	43%

Figure 4.7: File-level aggregate results.

the file-level aggregate results (Table 4.7) in order to compare them with a prior work. The aggregate results are computed by merging malware-detection results across independent executions of a malware detector. Note that we count each file exactly once; for example, if a malware detector detects the same malicious file over multiple NTWs, we count it only once.

False positive rate. The downloader detector [110] achieves the lowest FP rate. It raises 1.0% false positives on the set of downloaders; however, downloaders constitute only a small portion of the entire dataset (439K out of 20.55M files). Thus, its effective FP rate comes down to 0.021%, which is reached at the cost of excluding more than 20 million files (more than 97.3%) from the analysis. The other prior work – an LD [117] – has a fixed false-positive rate of 5%, which we set up in our experiments to achieve at least a 90% TP rate.

Surprisingly, the neighborhood detector's FP rate is only marginally worse

than the local detector's FP rate – 5.8% in comparison to 5%. However, it filters out significant portion of benign files, which helps Centurion to reduce the FP rate by 10.7× (5.8% vs. 0.54%) by using the neighborhood detector as a file filter. In comparison to the local detector, Centurion has a 9.3× lower FP rate (5% vs. 0.54%); thus it brings the absolute number of false positives from ∼1.2M down to ∼109.9K. Therefore, deeper (even human-level) analysis becomes feasible, i.e., ∼109.9K false alerts over a five-year period correspond to 60 false alerts per day on average.

True positive rate. The downloader detector [110] has the lowest TP rate due to its inherent inability to analyze non-downloaders. Therefore, it discovers 96% of malicious downloaders, but only 31.39% of all malware samples; it misses 94K out of 137K malware samples. Note that the Wine dataset may be skewed in favor of malicious downloaders; i.e., approximately one third of malware samples in the dataset are malicious downloaders. Thus, the downloader detector may have an even lower TP rate in a real deployment setting.

The neighborhood detector achieves a slightly lower TP rate (∼84%) because it erroneously filters out some malicious files, whereas LDs analyze all of them. Specifically, if the neighborhood detector fails to correlate malicious downloads appropriately, it may distribute malware samples across multiple predominantly benign neighborhoods. Due to low malware concentration, they may be excluded from the further analysis by the neighborhood classifier. The other reason why the neighborhood classifier misses some malware may be due to some labels incorrectly marking malicious domains as benign. Thus, malware samples

downloaded from such domains are excluded from further analysis.

In terms of true positives, Centurion inherits limitations of the neighborhood detector. It loses a few more percentage points due to running imperfect LDs within the neighborhoods that capture only 84% of malware, which results in a 78% TP rate. On the other hand, LDs demonstrate the highest TP rate (90.5%) because they are tuned to achieve a TP rate above 90%.

F-1 score. Centurion achieves the highest F-1 score (60.46%) because it detects a large portion of malware samples in the dataset ($\sim$78%) and maintains the low FP rate (0.54%). The next closest competitor – the downloader detector [110] – achieves only a 46.64% F-1 score due to its low TP rate. Interestingly, the LD [117] demonstrates $\sim$2.3 $\times$ worse results than the downloader detector because of the much higher FP rate.

4.4.6 Machine-level Aggregate Results

For completeness, we also describe machine-level detection results. However, we mainly focus on the new trends unobserved at the file level. We consider a machine to be compromised (or infected) if it has downloaded at least one malicious file. Though Centurion is meant to be a file-level detector rather than a machine-level detector, it achieves promising machine-level detection results.

False positive rate. In terms of false positives, we notice two opposing trends. First, the machine-level false-positive rate is higher than the file-level false-positive rate for all detectors, because a detector mislabeling a single benign file may dramatically affect the false-positive rate if the file has been downloaded on multiple

Parameter	Prior Work [CCS'15]	Local Detectors [DIMVA'16]	Neighborhood Detector	Centurion
Clean machines	3,753,931			
Compromised machines	768,525			
False Positives, machines	282,203	862,434	2,991,981	433,015
	7.51%	22.97%	79.70%	11.53%
True Positives (Recall)	98,112	670,164	739,558	651,025
	12.76%	87.20%	96.23%	84.71%
Precision	25.80%	43.73%	19.82%	60.06%
F-1 score	17.08%	58.25%	32.87%	70.28%
Time to Detection, days	84.26	0	20.63	28.67

Figure 4.8: Machine-level aggregate analysis.

clean machines – i.e., those machines become false positives.

Second, relative FP rates of all detectors significantly differ across file- and machine-level detection experiments. For example, the downloader detector's FP rate is only 1.53 times lower than Centurion's at the machine level, in comparison to a 7.1$\times$ difference at the file level. The downloader detector's results worsen mainly because the detector often mislabels benign files that are frequently downloaded on multiple clean machines, so those machines are considered false positives. Surprisingly, the neighborhood detector's FP rate reaches almost 80%, making it completely unusable. For this reason, we exclude it from further discussion.

True positive rate. In comparison to the FP rate, the TP rate does not exhibit a single trend; the direction in which it moves depends on a particular detector. The downloader detector's TP rate drops down by almost 3$\times$ because the majority of machines in the Wine dataset are infected by non-downloaders (malware that does

not download other files). As a result, the downloader detector misses almost 87% of infected machines.

Centurion's TP rate demonstrates the opposite trend. It increases file-level detection by 6.7% because it searches for correlated malicious downloads and thus is likely to detect similar malware that infects multiple machines. As a result, spatial correlations between malware downloads boost detection results – they raise from 78.03% up to 84.71%.

F-1 score. Overall, Centurion achieves the best FP/TP trade-off, having the highest F-1 score (60.06%). The downloader detector demonstrates the poorest detection results, having the lowest F-1 score (17.08%), mainly due to the low TP rate.

Time to detection. We observe that average time to detection slightly increases for Centurion (from 20.33 days to 28.67 days), but it is almost three times lower than the same parameter of the downloader detector because Centurion makes a decision regarding a file without waiting until it downloads other files.

4.5 Real-Time Detection

4.5.1 File-level real-time detection

We start with the analysis of the temporal distribution of the download events (Figure 4.9) to visualize file downloads over time. Every time Centurion runs, it analyzes download events within an NTW, which is set to 150 days in our experiments. Therefore, we represent the intensity of downloads over time as the number of downloads within each NTW. Specifically, for each time stamp, we com-

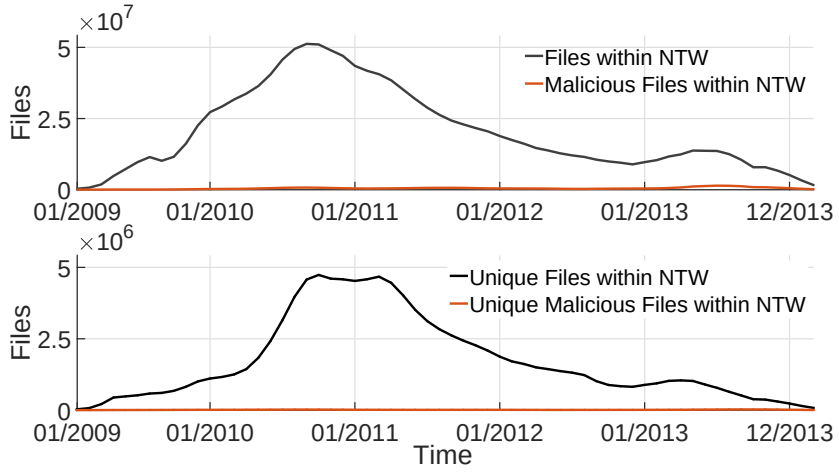

Figure 4.9: (File-level real-time detection) File download statistics.

pute the total number of file downloads and the number of malicious file downloads within the previous NTW (upper plot of Figure 4.9). For example, the value labeled as 01/2011 includes file downloads from 06/2010 until 01/2011. We also visualize the total number of distinct downloads and the number of distinct malicious downloads (lower plot of Figure 4.9).

Every point on these curves characterizes the number of files Centurion has to deal with when operating in a real-time detection mode. The large gap between the black and the red curves shows that only a small percentage of files in the Symantec Wine dataset is malicious. Centurion manages to filter out most benign files from further analysis and thus reduces the overall false-positive rate.

When taking a deeper look at the plots, we notice that file downloads in the Symantec Wine dataset exhibit a non-uniform pattern over time. The total number of downloads increased from January 2008 and reached its peak (51 million down-

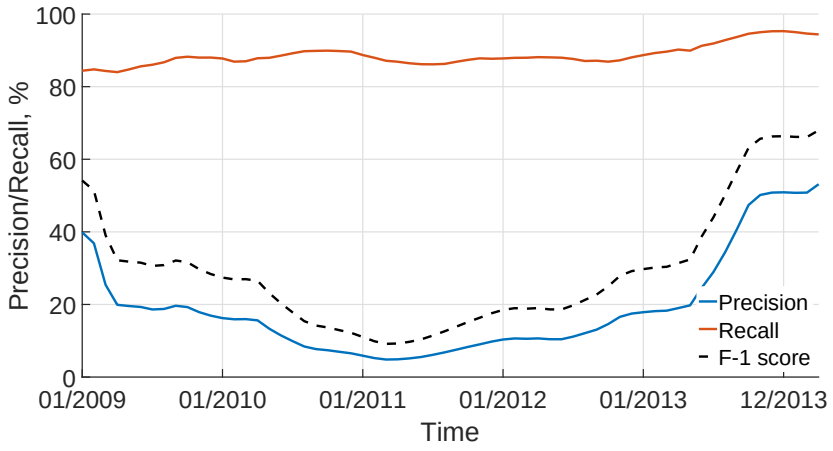

Figure 4.10: (File-level real-time detection) Local detector's detection results.

loads per NTW) within the NTW ending in October 2010. After that, it decreased over time. The temporal pattern of distinct downloads slightly differs: intensity of distinct downloads reached a flat plateau (4.74M per NTW) in September 2010 and remained on approximately the same level until April 2011. However, malicious files are responsible for only the small percentage of all downloads – at most 1.43M total malicious downloads and 27K unique malicious downloads.

Note that the distribution's low intensive ends impose an obstacle for Centurion because of the insufficient number of correlated file downloads. For this reason, we discard file downloads from before June 2008. Therefore, we run Centurion the first time on the neighborhood window spanning the interval from 06/2008 until 01/2009 and label results with the 01/2009 time stamp.

Local detectors. When we analyze the temporal behavior of local detectors (Figure 4.10), we notice an anti-correlation between the total number of unique down-

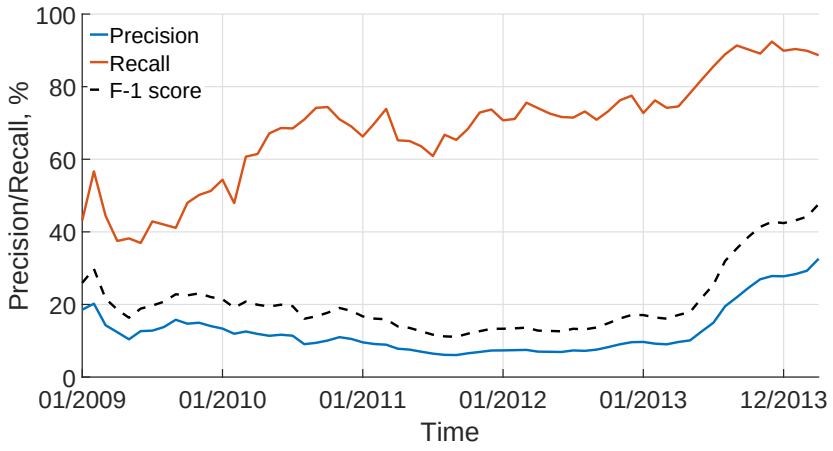

Figure 4.11: (File-level real-time detection) Neighborhood detector's detection results.

loads and LD precision. The peak of unique downloads corresponds to the large number of benign downloads. Therefore, when an LD processes them, it outputs a large number of false-positive alerts, which results in a precision drop (it drops to a level of less than 5%). However, the recall stays in the range of 84%–95% because it depends only on LDs' ability to detect malicious files. The F-1 scores lean more toward precision than recall; that is why the LD has mostly low F-1 scores over a large period of time (between 9% and 66%).

Neighborhood detector. Before analyzing Centurion's real-time detection, we briefly discuss the neighborhood detector's performance. We assume that the neighborhood detector (Figure 4.11) labels all the files within malicious neighborhoods as malicious. Both the LD and the neighborhood detector suffer from low precision. However, the underlying cause is different. The neighborhood classifier is supposed to label neighborhoods malicious if they comprise more than 5% malicious files.

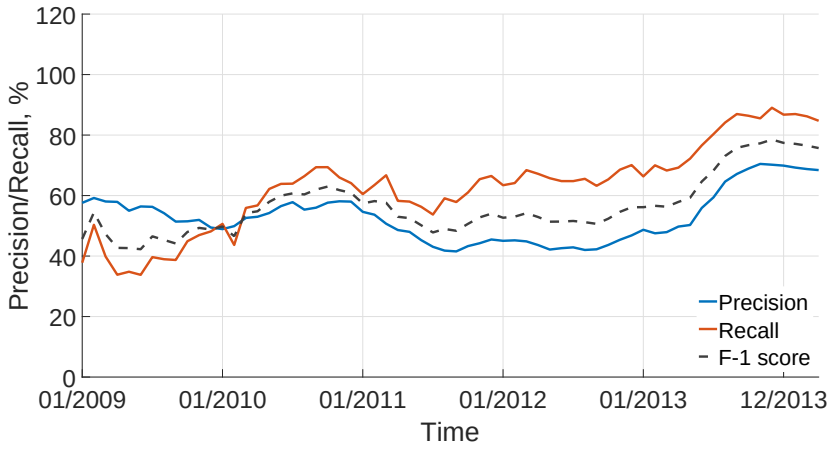

Figure 4.12: (File-level real-time detection) Centurion's detection results.

Usually, most files in a neighborhood are benign. Thus, when the neighborhood detector conservatively labels all the files malicious, it suffers from a high false-positive rate and, consequently, low precision. Hence, the neighborhood detector is designed to be conservative. Also the neighborhood detector inadvertently filters out some malicious files, which leads to results that are lower than LDs' recall.

Centurion. In comparison to the LDs, Centurion boosts precision and inherits a slightly lower recall from the neighborhood classifier because it only aggregates LD predictions collected across suspicious neighborhoods (Figure 4.12). Due to a low false-positive rate, Centurion achieves high precision – i.e., many benign files are filtered out by the neighborhood detector before the rest of them are reanalyzed by Centurion. Thus, LDs running within suspicious neighborhoods analyze fewer benign files than LDs in the traditional deployment scenario. At the same time, Centurion has a slightly lower recall than both LDs and the neighborhood detector because Centurion labels a file as malicious only if it is contained within a suspi-

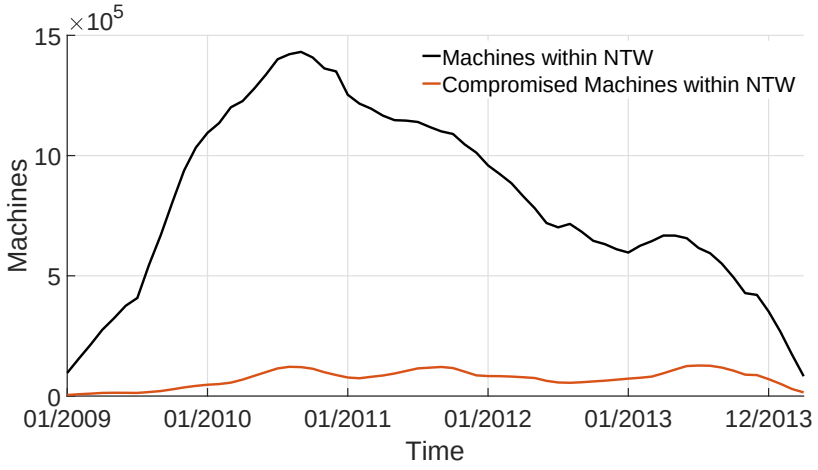

Figure 4.13: (Machine-level real-time detection) Machine-level local statistics.

cious neighborhood and an LD raises a file-level alert. However, neighborhood and domain name classifiers are imperfect – they may fail to correctly label malicious neighborhoods and domains, respectively. Therefore, Centurion does not aggregate LD output across all malicious files, which results in a slightly lower recall. Centurion's F-1 score is bounded by close values of precision and recall and is much higher than the same LD and neighborhood detector parameters. To quantitatively compare Centurion with local detectors we, compute the area under the F-1 curve. At the file level, Centurion achieves a 96.6% higher area under the F-1 curve than the LD.

4.5.2 Machine-level real-time detection

Machine-level statistics (Figure 4.13) are similar to file-level statistics: only a small percentage of machines is compromised within each NTW window. The number of machines and compromised machines reached their peak values of 1.43M

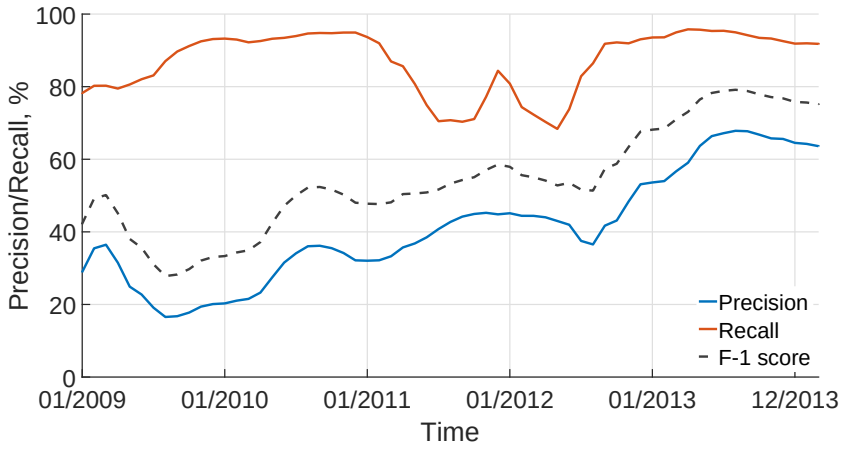

Figure 4.14: (Machine-level real-time detection) Machine-level local detector's detection results.

and 126.9K, respectively, in October 2010; i.e., at the peak, less than 8.9% of machines were compromised.

Overall, we observe higher values of precision and recall for all detectors (Figures 4.14, 4.15, 4.16) because, when interpreting detection results at the machine level, the detectors do not have to be very precise. They simply must detect at least one malicious file on an infected machine; file-level false positives on a particular machine do not count if that machine is infected.

Similar to file-level detection results, local detectors suffer from low precision because of the high number of false positives. However, precision is significantly higher – its mean value reaches 41%, as opposed to the mean value of 19% for file-level detection. Such a dramatic difference results from the file-level false positives on compromised machines that do not affect detectors' precision at the machine level. In both cases, the recall curve exhibits similar behaviors.

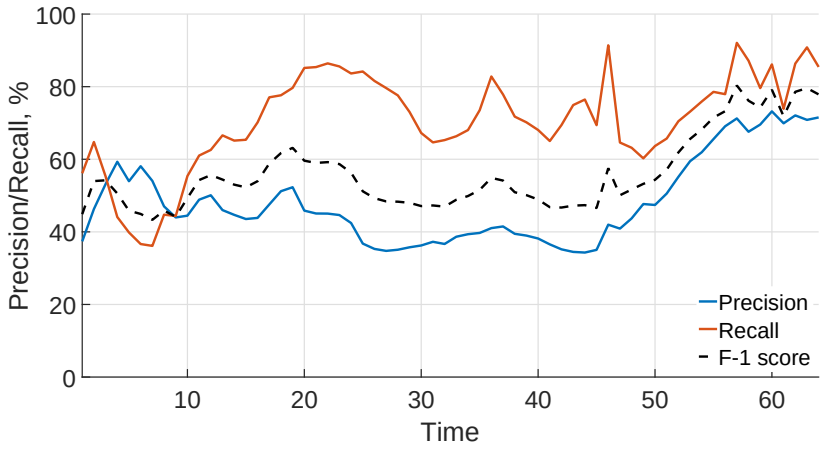

Figure 4.15: (Machine-level real-time detection) Machine-level neighborhood detector's detection results.

We observe a similar trend for the neighborhood detector – the mean precision value is 48%, as opposed to 12.5% at the file level. The recall values remain in the range of 36%–92%. Finally, Centurion brings the precision curve up at the cost of slightly lower recall. This is exactly the same effect that we observe at the file level.

Overall, Centurion achieves better results at the machine level than at the file level, which means that it can identify infected machines earlier and more robustly than individual malware samples. In the case of real-time detection, Centurion's main competitor is the LD. However, Centurion's area under F-1 curve is 28.6% higher than the similar parameter for the local detector.

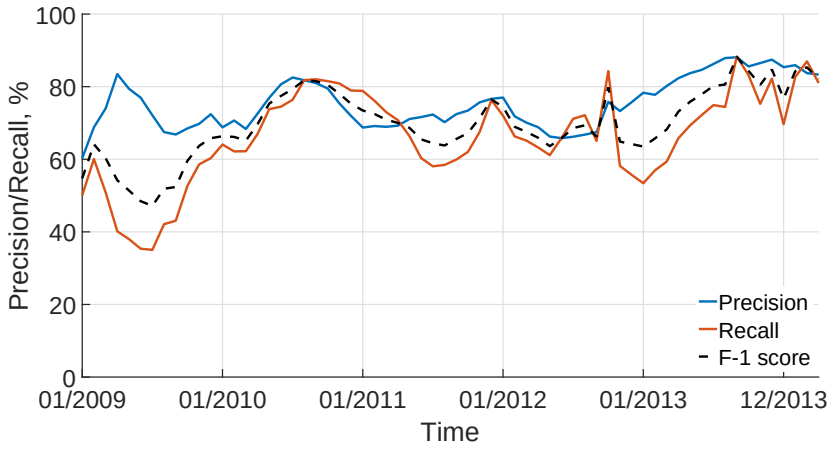

Figure 4.16: (Machine-level real-time detection) Machine-level Centurion's detection results.

4.5.3 Fragility of Count-GD

The traditional counterpart of Centurion is the Count-GD algorithm: to detect malware it counts the number of alerts over a neighborhood and compares it to a threshold. This threshold scales linearly with the size of the neighborhood. Next, we experimentally quantify the total error Count-GD can tolerate in the Symantec Wine setting (Figure 4.17). Note that the error in estimating neighborhood size can be double-sided: underestimates (negative errors) can make neighborhoods look like alert hotspots and lead to false positives, while overestimates (positive errors) can lead to missed detections (i.e., lower true positives).

We run Count-GD in the same setting as Centurion; i.e., we adjust Count-GD's threshold to match the performance of Centurion's neighborhood classifier (true-positive rate of 95.41%, Section 4.4.1) with zero neighborhood estimation errors (Figure 4.17).

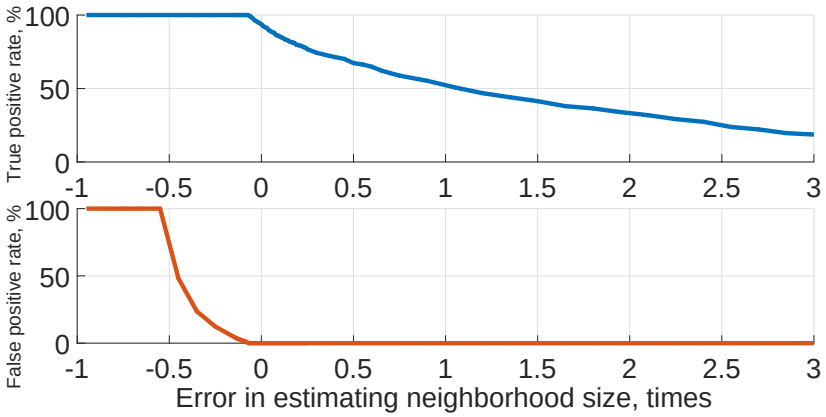

Figure 4.17: (Symantec Wine dataset) An error in estimating neighborhood size dramatically affects Count-GD's performance. Count-GD can tolerate at most 30% underestimation errors and 1% overestimation errors to achieve comparable with Shape GD performance.

Recall that in this setting the neighborhood classifier has a maximum global false-positive rate of 19.03% and a true-positive rate of 95.41%. For maintaining a similar performance for detection, our experiments show that Count-GD can only tolerate neighborhood size estimation errors within a very narrow range: $[-30\%, 1\%]$. A key takeaway here is that underestimating a neighborhood's size makes Count-GD extremely fragile (-30%). On the other hand, overestimating neighborhood sizes decreases true positives, and this effect is catastrophic.

Note the importance of this effect in practice. In the example of a Fortune 500 company, we observed that commercial Security Information and Event Management (SIEM) tools often do not report alerts in a timely manner and may delay delivering alerts by up to two months due to unpredictable infrastructure failures and a local IT service intervening in alert analysis. Also, given the practical de-

ployments where nodes become infected out of band (e.g., outside the corporate network) or go out of range (e.g., with mobile devices), the tight margins on errors can render Count-GD extremely unreliable. Even with sophisticated size-estimation algorithms, because the underlying distributions that create these neighborhoods (e.g. the number of clients per server) have sub-exponential heavy tails, such distributions typically result in poor parameter estimates due to lack of higher moments, and thus, poorer statistical concentrations of estimates of true value [81]. We see that by eliminating this size dependence of Count-GD, Centurion provides a robust inference algorithm.

4.6 Discussion

Evasion attacks. Centurion requires a human analyst to correctly specify attack vectors. If a new attack vector emerges (e.g., `badUSB`), then the corresponding attack may go undetected. However, attack vectors such as URLs or emails or physical devices along which malware propagates are far fewer than vulnerabilities, exploits, or malware samples. Thus, the majority of attackers keeps exploiting well-known attack vectors such as external URLs, email attachments, etc. Furthermore, individual LDs may be susceptible to evasion attacks, which may negatively affect Centurion's detection. However, designing evasion-resistant LDs [99] is outside the scope of this paper.

Detection results of Centurion vs. Shape-GD. In comparison to Shape-GD, which reduces the number of local false positives by $\sim$830 and $\sim$1000 times in watering hole and phishing experiments, respectively (Chapter 3), Centurion achieves only a

$\sim$9.3$\times$ reduction of false positives. There are multiple reasons why the reduction of the number of LD-level false positives differs dramatically across Shape-GD and Centurion frameworks though they share a similar theoretical foundation.

First, the Symantec Wine dataset contains incomplete information about malware spread. Specifically, Symantec's antivirus products are installed on a subset of machines targeted by attack, and some of those machines may not be susceptible to attack. As a result, the Symantec Wine dataset includes information about those machines that are monitored by Symantec's antivirus product and where the attack succeeded. However, Centurion needs to know all the machines that were targeted by a particular attack, such as with phishing and watering hole experiments. Partial knowledge makes it difficult for Centurion to find correlations between infected machines.

Second, the domain name classifier conservatively raises too many false alerts. Therefore, neighborhoods include many benign files. This issue can be further addressed by applying more advanced algorithms for establishing domain name reputation. Unfortunately, the Symantec Wine dataset lacks the necessary data (e.g., visited webpages, executed scripts on those webpages, etc.) in order to use more accurate algorithms.

Third, malware propagation in the Symantec Wine dataset happens at a very large scale; i.e., it takes days. However, Centurion is designed to detect 'bursty' fast-spreading malware campaigns (e.g., phishing and watering hole attacks). Centurion may end up merging multiple unrelated downloads into the same neighborhood, and many of them may be completely benign.

Finally, in watering hole and phishing experiments, correlations are more obvious for Shape-GD – malware spreads though URLs or email lists. Also, malware spreads much faster – within seconds in watering hole attacks and within few hours in phishing attacks.

4.7 Conclusion

Building robust behavioral detectors is a long-standing problem, especially in large distributed systems where false positives can be overwhelming. We observe that attacks on enterprise networks induce low-dimensional neighborhoods on otherwise high-dimensional feature vectors, but such neighborhoods are unpredictable and thus difficult to exploit. Centurion amplifies the malware signal through neighborhoods and exploits their shape to identify infected ones early. Automating the search for new neighborhoods – i.e., new attack vectors – that correlate with confirmed infections, would be a natural next step toward deployable behavioral detectors.

Chapter 5

Related Work

This section introduces some of the most commonly used approaches to malware detection. First, we discuss approaches to host-level malware detection and then we talk about approaches to global malware detection (collaborative intrusion detection systems – CIDS) that work at the network level and monitor the behavior of multiple devices.

In the case of host-level detectors, we follow the classification introduced in Chapter 1 and distinguish two types of malware detectors: static and dynamic. The former directly analyze static code properties, while the latter execute and analyze a program's interaction with its runtime environment.

5.1 Static Analysis

Static analysis tools comprise two complementary approaches: static code analysis and machine-learning approaches that use statically extracted program features.

5.1.1 Program Analysis

The dominant approach to malware detection that falls into this category is static information flow tracking (IFT). As we mentioned before (Chapter 1), sound static program analysis usually suffers from high false-positive rate. Here we primarily cite recent papers implementing IFT for Android applications because they complement Sherlock and represent an alternative approach to malware detection.

FlowDroid [56] performs inter-procedural context, flow, field, and object-sensitive IFT analysis of Android applications. It is built on top of a generic IFDS framework [131] for inter-procedural data flow analysis.

Unlike FlowDroid, DroidSafe [86] ignores flow sensitivity to make analysis more scalable. Compared to FlowDroid, it achieves higher precision because it includes an Android framework in the analysis.

CHEX [113] searches for component hijacking vulnerabilities in Android applications by modeling them from a data flow analysis perspective.

Apposcopy [78] primarily focuses on the identification of Android malware that steals private user information. It incorporates a high-level language for specifying semantic characteristics of malware families and a static analysis for deciding whether an Android application matches a malware signature.

Amandroid [150] is a general framework that performs flow and context-sensitive points-to analysis for all objects in an Android application, which can then be used for other specialized types of program analysis.

Feng et al. [136] suggest using statically identified information flows within Android applications as a permission mechanism.

5.1.2 Machine Learning

One of the popular approaches to malware detection is to extract syntactic features from potentially malicious files, convert them into feature vectors, and use a machine-learning classifier to determine whether a file is malicious or not.

Drebin [55] extracts static features from Android programs, such as permissions and API calls, and uses a support vector machine (SVM) to separate benign apps from malware.

DroidAPIMiner [52] extracts API features and applies a k-nearest neighbors (kNN) classifier to distinguish malware from benignware.

Gascon et al. [84] propose a method for malware detection based on efficient embeddings of function call graphs with an explicit feature map inspired by a linear-time graph kernel.

Peng et al. [128] introduce the notion of risk scoring and risk to reduce risk communication for Android applications. They identify three desiderata for an effective risk-scoring scheme. They propose using probabilistic generative models for risk-scoring schemes and identify several such models, ranging from the simple Naive Bayes to advanced hierarchical mixture models.

5.2 Dynamic Analysis

Dynamic analysis typically employs either dynamic information flow tracking or machine-learning methods that analyze a program's interaction with its runtime environment.

5.2.1 Program Analysis

The papers in this category mostly rely on dynamic information flow tracking to verify security properties of a system under test.

Dynamic information flow tracking analyzes how sensitive information is propagated through program code when the code is executed. In comparison to static IFT, dynamic IFT is not sound; that is why it is feasible to make it complete, i.e., it exploits a different design point in the soundness/completeness space.

Dynamic IFT systems require either virtualization or a special hardware. TaintDroid [75] and VetDroid [162] implement dynamic IFT by instrumenting Dalvik VM, the virtual machine used to run Android applications.

DroidScope [157] and its successor Decaf [93] bring dynamic instrumentation to a new level. The former modifies Android OSs and provides an API for implementing runtime analyses for security, while the latter makes localized modifications to the Qemu virtual machine [58] such that multiple OSs can be instrumented for security analyses, including information flow tracking.

Panorama [161] represents a suspicious application as a dynamic taint graph where nodes are OS objects and edges are information flows between them. The

system checks whether any of the provided data containment rules are violated within the graph.

Dytan [70], in addition to explicit information flows, also tracks implicit information flows within x86 executable files.

Saxeena et al. [133] and Chang et al. [63] rely on code rewriting to implement efficient IFT analyses. The former project works directly with binary code, while the latter transforms source code.

TriggerScope [82] employs static analysis to analyze path constraints guarding suspicious code blocks. If the valid input range for a path constraint is narrow, then the guarded code is considered malicious.

Guozhu et al. [115] developed a representation of Android malware families in the form of deterministic symbolic automata (DSA). They also propose an effective method of checking whether an Android application matches one or more previously learned DSA. If it does, then it is treated as malicious.

5.2.2 Machine Learning

System calls and middleware API calls have been studied extensively as a signal for behavioral detectors [57, 62, 67, 80, 83, 132, 149]. More recently, behavioral detectors have used signals such as power consumption[69], CPU utilization, memory footprint, and hardware performance counters [72, 143].

Detectors then extract features from these raw signals. For example, an *n*-gram is a contiguous sequence of n items that captures total order relations [62, 88],

n-tuples are ordered events that do not require contiguity, and bags are simply histograms. These can be combined to create bags of tuples, tuples of bags, and tuples of *n*-grams [62, 80] often using principal component analysis to reduce dimensions. Furthermore, system calls along with their arguments form a dependency graph structure that can be compared to sub-graphs that represent malicious behaviors [57, 67, 107].

Finally, detectors train models to classify executions into malware/benignware using supervised (signature-based) or unsupervised (anomaly-based) learning. These models range from distance metrics, histogram comparison, hidden Markov models (HMMs), and neural networks (artificial neural networks, fuzzy neural networks, etc.) to more common classifiers such as kNN, one-class SVMs, decision trees, and ensembles thereof.

Such machine-learning models, however, result in high false positives and negatives. Anomaly detectors can be circumvented by mimicry attacks where malware mimics system calls of benign applications [149] or hides within the diversity of benign network traffic[140]. Sommer et al. [140] additionally highlight several problems that can arise due to overfitting a model to a non-representative training set, suggesting signature-based detectors as the primary choice for real deployments. Unfortunately, signature-based detectors cannot detect new (zero-day) attacks. For Android, both system calls [61] and hardware-counter-based detectors [72] yield $\sim$20% false positives and $\sim$80% true positives.

Finally, with their ability to extract highly effective features, deep nets *may* provide a new way forward for creating novel behavioral detectors. At the global

level, however, what is needed is a data-light approach for global detection by composing LDs that are agile enough to perform global detection in a fast-changing (non-stationary) environment.

5.3 Collaborative Intrusion Detection Systems (CIDS)

Collaborative intrusion detection systems (CIDSs) provide an architecture where local detector (LD) alerts are aggregated by a global detector (GD). GDs can be either signature-based or anomaly-based [147, 163], or even a combination of the two [109] to generate global alerts. Additionally, the CIDS architecture can be centralized, hierarchical, or distributed (using a peer-to-peer overlay network) [163].

In all cases, existing GDs use some variant of either clustering- or count-based algorithms to aggregate LD alerts. Count-based GDs raise an alert once the number of alerts exceeds a threshold within a space–time window, while clustering-based GDs apply heuristics to control the number of alerts [71, 88, 89, 137, 154]. In HIDE [163], the GD at each hierarchical tier is a neural network trained on network traffic information. Worminator[112] additionally uses bloom filters to compact LDs' outputs and schedules LDs to form groups in order to spread alert information quickly through a distributed system. All count- and clustering-based algorithms are fragile when the noise is high (in the early stages of an infection) and when the network size is uncertain. In contrast, Shape-GD and Centurion are robust against such uncertainty.

Note that distributed CIDSs are vulnerable to probe-response attacks, where the attacker probes the network to find the location and defensive capabilities of a

local detector [60, 138, 139].

Chapter 6

Conclusions and Future Work

6.1 Thesis Contributions

This thesis presents an end-to-end behavioral malware detector, Shape-GD. Unlike traditional approaches to computer security (e.g., OS confinement, program analysis, etc.), which often fail to mitigate modern attacks (e.g., watering hole attacks, phishing attacks, or attacks that abuse hardware properties such as Spectre, Meltdown, and Rowhammer), Shape-GD can detect a broad class of computer security attacks early and robustly.

The Sherlock project describes a new design for a lightweight malware detector that employs machine-learning techniques to analyze dynamic instruction traces, along with architectural and micro-architectural states. It can therefore observe instruction-level behaviors that exploit the gap between the system's software abstractions and hardware implementations. Sherlock's design is especially applicable to resource-constrained devices, such as mobile phones and IoT devices, against diverse malware types. Furthermore, it achieves small TCBs and low overhead costs.

The Sherlock project introduces a new white-box methodology together with an operating range concept to evaluate malware detectors against evasive malware.

Its operating range concept is a step toward explainability of machine-learning-based malware detectors. Using these principles, we were able to build a detector that outperforms a prior work by 12.5% – 24.7% (AUC metric) and demonstrate that malware that attempts to evade static program analysis by adding run-time obfuscation appears more anomalous to Sherlock.

Our Shape-GD malware detector can compose Sherlock-like detectors. Shape-GD is a global detector that relies on two fundamental observations. First, attacks spread through a limited number of well-known attack vectors (e.g., compromised URLs or emails with malicious attachments). Shape-GD partitions devices (e.g., desktops, mobile and IoT devices) within a network into *neighborhoods*. Devices within a neighborhood are likely to be exposed to similar attack vectors. Second, the distributional shape of false positives is different from that of true positives. Although this difference is impossible for LDs to exploit, Shape-GD can aggregate alert-inducing (i.e., suspicious) feature vectors from a neighborhood to classify whether these feature vectors are drawn from a true-positive distribution. Experiments demonstrate that Shape-GD identifies malware early ($\sim$100 infected nodes in a $\sim$100K-node system for watering hole attacks and $\sim$10 of 1000 for phishing attacks) and robustly (with $\sim$100% global true-positive and $\sim$1% global false-positive rates).

Finally, we present the Centurion malware detector, which is designed to detect malware among Symantec's clients. Though Centurion shares a common theoretical foundation with the Shape-GD detector, it significantly improves upon Shape-GD's design to accommodate the Symantec dataset's constraints. We evalu-

ate Centurion with 5 years of logs of 5 million Symantec client devices and show that Centurion is able to efficiently discover malware in real time by correlating file downloads across multiple machines. Specifically, it reduces false positives from $\sim$1M to $\sim$110K in comparison to a recent local detector and increases the true-positive rate by $\sim$2.5 times in comparison to a recent detector that analyzes metadata associated with file downloads. In addition, Centurion detects malware an average of 345 days earlier than commercial antivirus products.

6.2 Future Work

Here we outline several possible extensions for the projects described in the thesis.

Identify which attack classes Sherlock can best detect. One potential direction to extend Sherlock is an empirical study of attacks that can be efficiently detected by low-level detectors similar to Sherlock. We have achieved encouraging results for the following attacks: JIT spray, RowHammer [105, 135], microarchitectural covert channels [97], a side channel through a floating point unit [54], and an attack bypassing kernel address space layout randomization [87].

Apply Sherlock's methodology to other types of detectors. Shape-GD's white-box methodology together with its operating range can be applied to other types of malware detectors; its applicability is not limited to low-level detectors. We are currently generalizing the white-box methodology and applying it to a detector [110] that identifies malicious downloaders in the Symantec Wine dataset.

Learn neighborhood templates. Both Shape-GD and Centurion rely on templates that define common attack vectors. Those templates are currently provided by human analysts. However, both malware detectors would benefit from a data-driven approach to establishing such templates using labeled data, which would reduce the risk of human error.

Another research direction would be to learn a low-dimensional basis used by Shape-GD and Centurion to build vector histograms to increase distinguishability of false- and true-positive distributions (the shape property) at the neighborhood level.

Extend behavioral software analysis beyond computer security. We are currently developing a machine-learning framework to detect transient performance bugs that cause abrupt performance degradation. This framework monitors systems for anomalous behavior and identifies parts of the system that might be root causes of the observed anomalies. Specifically, the algorithm learns a complex system's regular dynamic behavior, which it uses to distinguish short-lived deviations (anomalies) from regular behavior, and then raises an alert when an anomaly is found. Another class of algorithms analyzes anomalies, filters out non-performance-related anomalies, and pinpoints quality-of-service-related metrics (such as jank) that are hard to detect and quantify without our anomaly detector.

Bibliography

[1] "Advanced malware protection (amp),"
http://www.cisco.com/c/en/us/products/security/advanced-malware-protection/index.html.

[2] "Advanced malware protection and detection (ampd),"
https://www.secureworks.com/capabilities/managed-security/network-security/advanced-malware-protection.

[3] "Android obad technical analysis paper, comodo malware analysis team,"
https://www.comodo.com/resources/Android_OBAD_Tech_Reportv3.pdf.

[4] "Android rat malware,"
http://www.itpro.co.uk/malware/22627/android-rat-malware-invades-mobile-banking-apps.

[5] "Applock vulnerability,"
http://blog.trendmicro.com/trendlabs-security-intelligence/applock-vulnerability-leaves-configuration-files-open-for-exploit.

[6] "Attack graphs: visualizing 200m alerts a day."
http://on-demand.gputechconf.com/gtc/2016/presentation/s6114-leo-meyerovich-attack-graphs-visualizing-alerts.pdf.

[7] “The bitfinex bitcoin hack: what we know (and don’t know),”
https://www.coindesk.com/bitfinex-bitcoin-hack-know-dont-knowl.

[8] “Data binning,”
https://en.wikipedia.org/wiki/Histogram.

[9] “Dissecting android’s bouncer,”
https://jon.oberheide.org/files/summercon12-bouncer.pdf.

[10] “Enron email dataset,”
https://www.cs.cmu.edu/~./enron.

[11] “Eset: En route with sednit,”
https://www.welivesecurity.com/wp-content/uploads/2016/10/eset-sednit-part1.pdf.

[12] “Evernote patches,”
http://blog.trendmicro.com/trendlabs-security-intelligence/evernote-patches-vulnerability-in-android-app/.

[13] “Exploid,”
http://forum.xda-developers.com/showthread.php?t=739874.

[14] “G4 - yahoo! network flows data,”
https://webscope.sandbox.yahoo.com.

[15] “Geinimi malware,”
https://nakedsecurity.sophos.com/2010/12/31/geinimi-android-trojan-horse-discovered/.

[16] “Gingerbreak apk root,”
http://droidmodderx.com/gingerbreak-apk-rootyour-gingerbread-device.

[17] “Google proguard,”
http://developer.android.com/tools/help/proguard.html.

[18] “Graphistry,”
https://www.iqt.org/graphistry.

[19] “Grr rapid response: remote live forensics for incident response,”
https://github.com/google/grr.

[20] “Hone tool,”
https://github.com/pmoody-/Linux-Sensor.

[21] “Inside the cyberattack that shocked the us government,”
https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government.

[22] “Intrusion into the dnc in 2016,”
https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee.

[23] “Jitbit macro recorder,”
http://www.jitbit.com.

[24] “Kaspersky security bulletin 2015,”
https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf.

[25] "Lastline: Defeat advanced malware before it infiltrates your network,"
https://www.lastline.com.

[26] "The latest phishing activity,"
http://www.fraudwatchinternational.com/phishing-alerts.

[27] "Malware database,"
http://malware.lu.

[28] "Malware database,"
http://virusshare.com.

[29] "Malware statistics, visited on 07-23-2017,"
https://www.av-test.org/en/statistics/malware.

[30] "Master key vulnerability,"
http://blog.trendmicro.com/trendlabs-securityintelligence/trend-micro-solution-for-vulnerability-affecting-nearly-allandroid-devices.

[31] "Meltdown,"
https://arxiv.org/pdf/1801.01207.pdf.

[32] "Mobile bitcoin miner,"
https://blog.lookout.com/blog/2014/04/24/badlepricon-bitcoin.

[33] "Mobile malware database,"
http://contagiominidump.blogspot.com.

[34] “osquery – performant endpoint visibility,”
https://osquery.io/.

[35] “Securerank algorithm,”
https://blog.opendns.com/2013/03/28/secure-rank-a-large-scale-discovery-algorithm-for-predictive-detection.

[36] “Slow loris attack,” http://www.slashroot.in/slowloris-http-dosdenial-serviceattack-and-prevention.

[37] “Spectre attacks: Exploiting speculative execution,”
https://spectreattack.com/spectre.pdf.

[38] “Statement regarding cyber attack against anthem,”
http://www.sophos.com/en-us/threat-center/mobile-security-threat-report.aspx.

[39] “Symantec intelligence report,”
https://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence-report-01-2015-en-us.pdf.

[40] “Symantec report on black vine espionage group,”
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-black-vine-cyberespionage-group.pdf.

[41] “Target data breach in 2013,”
https://www.thesslstore.com/blog/2013-target-data-breach-settled.

[42] "Trendlabs a look at google bouncer,"
http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer.

[43] "Ui/application exerciser monkey,"
http://developer.android.com/tools/help/monkey.html.

[44] "Universal android rooting procedure (rage method),"
http://theunlockr.com/2010/10/26/universal-android-rooting-procedure-rage-method.

[45] "An unprecedented look at stuxnet, the world's first digital weapon,"
https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet.

[46] "Virustotal – free online virus, malware and url scanner,"
https://www.virustotal.com.

[47] "Vulnerable & aggressive adware,"
http://www.fireeye.com/blog/technical/2013/10/ad-vulna-a-vulnaggressive-vulnerable-aggressive-adware-threatening-millions.html.

[48] "Wasserstein metric,"
https://en.wikipedia.org/wiki/Wasserstein_metric.

[49] "Why watering hole attacks work,"
https://threatpost.com/why-watering-hole-attacks-work-032013/77647.

[50] "Worldwide intelligence network environment," http://www.symantec.com/about/profile/universityresearch/sharing.jsp.

[51] "Yahoo data breach in 2014," https://www.nytimes.com/2017/03/17/technology/yahoo-hack-data-indictments.html.

[52] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, 2013, pp. 86–103.

[53] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," ser. HASP '13, 2013.

[54] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing." in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015.

[55] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," 2014.

[56] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings*

of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014.

[57] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Recent Advances in Intrusion Detection*, 2007.

[58] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[59] J. Benamou and Y. Brenier, "Mixed l2-wasserstein optimal mapping between prescribed density functions," *Journal of Optimization Theory and Applications*, 2001.

[60] J. Bethencourt, J. Franklin, and M. Vernon, "Mapping internet sensors with probe response attacks," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[61] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM, 2011.

[62] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in

Proceedings of the 2012 International Symposium on Software Testing and Analysis, ser. ISSTA 2012, 2012.

[63] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 39–50.

[64] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016.

[65] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng, "Evolutionary spectral clustering by incorporating temporal smoothness," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07, 2007.

[66] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications."

[67] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 1st India Software Engineering Conference*, ser. ISEC, 2008.

[68] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

[69] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "WattsUpDoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *USENIX Workshop on Health Information Technologies*, 2013.

[70] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 196–206.

[71] D. Dash, B. Kveton, J. M. Agosta, E. Schooler, J. Chandrashekar, A. Bachrach, and A. Newman, "When gossip is good: Distributed probabilistic inference for detection of slow network intrusions," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, ser. AAAI'06. AAAI Press, 2006, pp. 1115–1122.

[72] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 559–570.

[73] D. L. Donoho and P. J. Huber, "The notion of breakdown point," *A festschrift for Erich L. Lehmann*, vol. 157184, 1983.

[74] T. Dumitras and D. Shou, "Toward a standard benchmark for computer security research: The worldwide intelligence network environment (wine)," in

Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, ser. BADGERS '11, 2011.

[75] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010.

[76] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *CCS2011*, CCS2011addr, CCS2011date 2011, pp. 627–638.

[77] A. P. Felt and D. Wagner, "Phishing on mobile devices," in *In W2SP*, 2011.

[78] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 576–587.

[79] G. A. Fink, V. Duggirala, R. Correa, and C. North, "Bridging the host-network divide: Survey, taxonomy, and solution," in *Proceedings of the 20th Conference on Large Installation System Administration*, ser. LISA '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 20–20.

[80] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for

unix processes," in *Security and Privacy, 1996. Proceedings., IEEE Symposium on*, 1996.

[81] S. Foss, D. Korshunov, and S. Zachary, "An introduction to heavy-tailed and subexponential distributions," 2009, springer Series in Operations Research and Financial Engineering.

[82] Y. Fratantonio, A. Bianchi, W. K. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 377–396.

[83] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *IEEE Symposium on Security and Privacy*, 2010.

[84] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, ser. AISec '13. New York, NY, USA: ACM, 2013, pp. 45–54.

[85] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 72–81.

[86] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, , and M. Rinard, "Information-flow analysis of android applications in droidsafe," in *In Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[87] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr." ser. CCS '16, 2016.

[88] G. Gu, J. Zhang, and W. Lee, "Botsniffer: Detecting botnet command and control channels in network traffic," in *Presented at the 16th Annual Network & Distributed System Security Symposium*. NDSS, 2008.

[89] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "Bothunter: Detecting malware infection through ids-driven dialog correlation," in *Proceedings of 16th USENIX Security Symposium*, 2007.

[90] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001.

[91] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[92] S. Hao, A. Kantchelian, B. Miller, V. Paxson, and N. Feamster, "Predator: Proactive recognition and elimination of domain abuse at

time-of-registration," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1568–1579. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978317

[93] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 248–258.

[94] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:1.

[95] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin, "Mutantx-s: Scalable malware clustering based on static features," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 187–198.

[96] P. J. Huber, *Robust statistics*. Springer, 2011.

[97] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-driven covert channels and using them for

defense," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2015.

[98] L. Invernizzi, S. Miskovic, R. Torres, C. Kruegel, S. Saha, G. Vigna, S. Lee, and M. Mellia, "Nazca: Detecting malware distribution in large-scale networks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[99] A. Kantchelian, "Taming Evasions in Machine Learning Based Detection Pipelines," Ph.D. dissertation, 2016. [Online]. Available: http://www.escholarship.org/uc/item/1n7599wp

[100] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley, 1990.

[101] M. Kazdagli, L. Huang, V. Reddi, and M. Tiwari, "Morpheus: Benchmarking computational diversity in mobile malware," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:8.

[102] M. Kazdagli, V. J. Reddi, and M. Tiwari, "Quantifying and improving the efficiency of hardware-based mobile malware detectors," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 37:1–37:13.

[103] K. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware de-

tection," in *18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[104] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *Research in Attacks, Intrusions, and Defenses*. Springer International Publishing, 2015, pp. 3–25.

[105] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 361–372.

[106] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[107] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[108] F. Kooti, L. M. Aiello, M. Grbovic, K. Lerman, and A. Mantrach, "Evolution of conversations in the age of email overload," in *Proceedings of 24th International World Wide Web Conferenced (WWW)*, 2015.

[109] C. Krüegel, T. Toth, and C. Kerer, "Decentralized event correlation for intrusion detection," in *Proceedings of the 4th International Conference Seoul on Information Security and Cryptology*, ser. ICISC '01. Springer-Verlag, 2002.

[110] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitras, "The dropper effect: Insights into malware distribution with downloader graph analytics," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1118–1129.

[111] B. J. Kwon, V. Srinivas, A. Deshpande, and T. Dumitras, "Catching worms, trojan horses and pups: Unsupervised detection of silent delivery campaigns," in *NDSS*, 2017.

[112] M. Locasto, J. Parekh, A. Keromytis, and S. Stolfo, "Towards collaborative security and p2p intrusion detection," in *Information Assurance Workshop, IAW*, 2005.

[113] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 229–240.

[114] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on*

Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234.

[115] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 306–317.

[116] S. J. Mihai Christodorescu, "Static analysis of executables to detect malicious patterns," The University of Wisconsin, Madison, Tech. Rep., 2006.

[117] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, A. D. Joseph, and J. D. Tygar, "Reviewer integration and performance measurement for malware detection," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 122–141.

[118] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov, "Botgrep: Finding p2p bots with structured graph analysis," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7.

[119] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and

an algorithm," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, 2001.

[120] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin, and S. Alrwais, "Detection of early-stage enterprise infection by mining large-scale log data," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 45–56.

[121] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1406–1418.

[122] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proceeding of the 21st International Symposium on High Performance Computer Architecture*, 2015.

[123] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 506–519. [Online]. Available: http://doi.acm.org/10.1145/3052973.3053009

[124] V. Paxson, “Bro: A system for detecting network intruders in real-time,” *Comput. Netw.*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[125] M. Payer, “Hexpads: A platform to detect ”stealth” attacks,” in *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, 2016, pp. 138–154.

[126] N. Peiravian and X. Zhu, “Machine learning for android malware detection using permission and api calls,” in *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, ser. ICTAI ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 300–305.

[127] D. Pelleg and A. W. Moore, “X-means: Extending k-means with efficient estimation of the number of clusters,” in *Proceedings of the 7th International Conference on Machine Learning*, 2000.

[128] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Using probabilistic generative models for ranking risks of android apps,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 241–252.

[129] D. Plohmann, K. Yakdan, M. Klatt, J. Bader, and E. Gerhards-Padilla, “A comprehensive measurement study of domain generating malware,” in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 263–278.

[130] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–15.

[131] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61.

[132] W. Robertson, F. Maggi, C. Kruegel, and G. Vigna, "Effective anomaly detection with scarce training data," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[133] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 74–83.

[134] R. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*. MIT Press, 2012.

[135] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," in *BlackHat*, 2015.

[136] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proceedings of the 29th ACM/IEEE International Conference on Automated*

Software Engineering, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 515–526.

[137] S. Shin, Z. Xu, and G. Gu, "EFFORT: Efficient and Effective Bot Malware Detection," in *Proceedings of the 31th Annual IEEE Conference on Computer Communications (INFOCOM'12) Mini-Conference*, March 2012.

[138] Y. Shinoda, K. Ikai, and M. Itoh, "Vulnerabilities of passive internet threat monitors," in *Proceedings of the 14th Conference on USENIX Security Symposium*, 2005.

[139] V. Shmatikov and M.-H. Wang, "Security against probe-response attacks in collaborative intrusion detection," in *the Workshop on Large Scale Attack Defense*, 2007.

[140] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *the IEEE Symposium on Security and Privacy*, 2010.

[141] N. Šrndić and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *the IEEE Symposium on Security and Privacy*, 2014.

[142] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, 2014, pp. 109–129.

[143] A. Tang, S. Sethumadhavan, and S. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses*, 2014.

[144] S. Vallender, "Calculation of the wasserstein distance between probability distributions on the line," *Theory of Probability & Its Applications*, vol. 18, no. 4, pp. 784–786, 1974.

[145] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016.

[146] E. Vasilomanolakis, S. Karuppayah, M. Muhlhauser, and M. Fischer, "Taxonomy and survey of collaborative intrusion detection," *ACM Comput. Surv.*, 2015.

[147] V. Vlachos, S. Androutsellis-Theotokis, and D. Spinellis, "Security applications of peer-to-peer networks," *Comput. Netw.*, vol. 45, no. 2, 2004.

[148] U. von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, 2007.

[149] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *the ACM Conference on Computer and Communications Security*, 2002.

[150] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1329–1341.

[151] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 1113–1120. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553516

[152] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[153] Y. Xie, H.-A. Kim, D. R. O'Hallaron, M. K. Reiter, and H. Zhang, "Seurat: A pointillist approach to anomaly detection," in *The International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2004.

[154] Y. Xie, H. Kim, D. R. O'Hallaron, M. K. Reiter, and H. Zhang, "Seurat: A pointillist approach to anomaly detection," in *Recent Advances in Intrusion Detection*, 2004.

[155] H. Xu, C. Caramanis, and S. Mannor, "Outlier-robust pca: the high-dimensional case," *IEEE transactions on information theory*, vol. 59, no. 1, pp. 546–572, 2013.

[156] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers: A case study on pdf malware classifiers," in *Network and Distributed Systems Symposium*, 2016.

[157] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29.

[158] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *Proceeding SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2016.

[159] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671271

[160] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. ACM, 2013, pp. 199–208.

[161] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing

system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 116–127.

[162] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. ACM, 2013, pp. 611–622.

[163] Z. Zhang, J. Li, C. N. Manikopoulos, J. Jorgenson, and J. Ucles, "Hide: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification," in *the IEEE Workshop on Information Assurance and Security*, 2001.

[164] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum, "Botgraph: Large scale spamming botnet detection," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 321–334.

[165] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Computers and Security*, vol. 29, no. 1, pp. 124 – 140, 2010.

[166] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceeding SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

Vita

Mikhail Kazdagli was born in Moscow, Russia. He received the Bachelor of Science degree and the Master of Science in Computer Science and Applied Mathematics from Moscow Institute of Physics and Technology and the Master of Science in Computer Science from Boston University. He applied to the University of Texas at Austin for enrollment in their software engineering program. He was accepted and started graduate studies in June, 2013.

Email: mikhail.kazdagli@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.