Detection and Classification of Malicious Processes Using System Call Analysis

A Thesis

Submitted to the Faculty

of

Drexel University

by

Raymond J. Canzanese, Jr.

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

May 2015

Dedications

To Madison,

without whose support and companionship this would

have been a far less rewarding and enjoyable experience.

Acknowledgments

Throughout the course of writing this thesis, I was surrounded by an outstanding community who provided the support, inspiration, encouragement, and distraction necessary for its completion. To my advisers, Moshe Kam and Spiros Mancoridis, and committee, Naga Kandasamy, Ko Nishino, Harish Sethu, and Steven Weber, I am most thankful. It was their insights, guidance, feedback, and support that made this thesis possible. I cannot overstate the depth and breadth of the lessons I have learned from them, which have shaped not only this thesis, but my general approach to problem solving, my outlook on life, and my ambitions.

That my advisers managed to find the time, energy, and patience to meet with me regularly to discuss my successes and failures confounds me. Watching them advance in their careers while remaining so humble and grounded has been truly inspiring. I also had the great fortune to learn from a number of other faculty members here at Drexel who have helped shape and inspire this work, including Kapil Dandekar, John Walsh, Tom Chmielewski, Ali Shokoufandeh, and Marcello Balduccini.

My friends and colleagues here at Drexel, especially those members of the Data Fusion Lab and Software Engineering Research Group, have also served to educate, motivate, and inspire me. Among these are Bradford Boyle, Jeff Wildman, Chris Lester, Sayandeep Acharya, Gus Anderson, Dave Dorsey, Pramod Abichandani, Rich Primerano, George Sworo, Alex Fridman, DJ Bucci, Yifei Xu, Feiyu Xiong, Ji Wang, Cole Gindhart, Arjun Rajasekar, Maxim Shevertalov, Ed Stehle, Rob Lange, Kevin Lynch, Bill Mongan, Bander Alsulamy, Matt Ping, and Alex Duff.

I am grateful for the years of support and assistance provided by the ECE staff, especially Chad Morris, Kathy Bryant, Tanita Chapelle, Phyllis D. Watson, Sean Clark, Taif Choudhury, and Wade Kirkpatrick.

I am especially grateful to the KEYSPOT Network, The City of Philadelphia's Office of Innovation and Technology (OIT), The Mayor's Commission on Literacy, and the Department of Parks and Recreation (PPR) for their support of their research. Thank you to the KEYSPOT Executive Committee – Judith Rényi, Patrick Morgan, Andrew Buss, and Joann Ferroni – for their support. Thank you also to Jennifer Kobrin, Maria Walker, Adena Klem, and all of those who helped make the KEYSPOT collaboration possible. None of this would have been possible without the selfless work of Andy Stutzman who facilitated this collaboration. Thank you to the People's Emergency Center, especially Tan Vu and Desmond Hinkle, and the Dornsife Center for Neighborhood Partnerships, especially Sarah Steltz and Kevin Williams, for their support and participation. Thank you also to Paul Dougherty and George Lauro, who afforded me the opportunity to share my work with a much wider audience than would have otherwise been possible.

Finally, thank you to my friends and family, without whose love, support, encouragement, enthusiasm, and distraction, none of this would have been possible.

This research is sponsored by a Secure and Trustworthy Cyberspace (SaTC) award from the National Science Foundation (NSF), under grant CNS-1228847.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ADD	average detection delay
API	application programming interface
APT	advanced persistent threat
AUC	area under the curve
AV	antivirus
CART	classification and regression trees
CUSUM	cumulative sum
DDoS	distributed denial of service
DoS	denial of service
ETW	Event Tracing for Windows
FPR	false positive rate
i.i.d.	independent, identically distributed
I/O	input/output
IDF	inverse document frequency
IDS	intrusion detection system
LDA	linear discriminant analysis
LLRT	log-likelihood ratio test
LPC	local procedure call
LR	logistic regression
LSA	latent semantic analysis
NIDS	network intrusion detection system
OS	operating system
OVA	one-versus-all
PCA	principal component analysis
PE	portable executable

PID	process identifier
PWS	password stealer
r.v.	random variable
RFE	recursive feature elimination
ROC	receiver operating characteristic
SCS	System Call Service
SEM	standard error of the mean
SGD	stochastic gradient descent
SPRT	sequential probability ratio test
SSDT	system service descriptor table
SVD	singular value decomposition
SVM	support vector machine
TF	term frequency
TF-IDF	term frequency – inverse document frequency
TID	thread identifier
TPR	true positive rate
ULS	ultra large scale
VM	virtual machine
VMM	virtual machine manager

List of Notation

$\{\cdot\}$	An unordered set
$\langle\cdot\rangle$	An ordered sequence
$[\cdot]$	A matrix or column vector
$\|\cdot\|$	The norm of a vector
$\mathbf{card}(\cdot)$	The cardinality of a set
$\odot$	Hadamard element-wise matrix product
$p_X(x)$	The probability distribution function of a random variable X
K	Number of malware classes
N	Number of training samples
l	System call trace length
n	Length of system call n-gram
$\mathcal{C}_k$	The set of all malware samples belonging to the k^{th} class
$\mathbf{y}$	A vector of binary labels
$\hat{\mathbf{x}}$	The feature vector of a system call trace output by the feature extractor
$\mathcal{X}$	The set of all training data
$\mathcal{X}_{\text{B}}$	The set of all training data collected from benign processes
$\mathcal{X}_{\text{M}}$	The set of all training data collected from malicious processes
X	A matrix formed by horizontally concatenating all training vectors
Λ	The decision statistic of a detector
λ	The decision threshold of a detector
$\mathbf{w}$	Vector of feature weights used by a detector
μ	Sample mean
σ^2	Sample standard deviation
σ	Sample variance

Abstract

Detection and Classification of Malicious Processes Using System Call Analysis
Raymond J. Canzanese, Jr.
Moshe Kam, Ph.D. and Spiros Mancoridis, Ph.D.

Despite efforts to mitigate the malware threat, the proliferation of malware continues, with record-setting numbers of malware samples being discovered each quarter. Malware are any intentionally malicious software, including software designed for extortion, sabotage, and espionage. Traditional malware defenses are primarily signature-based and heuristic-based, and include firewalls, intrusion detection systems, and antivirus software. Such defenses are reactive, performing well against known threats but struggling against new malware variants and zero-day threats. Together, the reactive nature of traditional defenses and the continuing spread of malware motivate the development of new techniques to detect such threats. One promising set of techniques uses features extracted from system call traces to infer malicious behaviors.

This thesis studies the problem of detecting and classifying malicious processes using system call trace analysis. The goal of this study is to identify techniques that are 'lightweight' enough and exhibit a low enough false positive rate to be deployed in production environments. The major contributions of this work are (1) a study of the effects of feature extraction strategy on malware detection performance; (2) the comparison of signature-based and statistical analysis techniques for malware detection and classification; (3) the use of sequential detection techniques to identify malicious behaviors as quickly as possible; (4) a study of malware detection performance at very low false positive rates; and (5) an extensive empirical evaluation, wherein the performance of the malware detection and classification systems are evaluated against data collected from production hosts and from the execution of recently discovered malware samples. The outcome of this study is a proof-of-concept system that detects the execution of malicious processes in production environments and classifies them according to their similarity to known malware.

Chapter 1: Introduction

Malware, short for 'malicious software', are any software with an intentionally malicious function. Malware are used for espionage, extortion, and sabotage, and used to perform other unauthorized tasks. Such tasks include sending unwanted email messages (spam) and participating in distributed denial of service (DDoS) attacks. Malware exist in various forms, including embedded malware, mobile malware, malicious scripts, and malicious bytecode. This study seeks to identify the execution of standalone binary executable malware files (*i.e.*, malware samples) in traditional computing environments (*e.g.*, servers, workstations, laptops, and notebooks). A computer host is considered 'infected' if it has executed any known malware samples. Otherwise, it is considered 'clean'.

In this chapter, Section 1.1 provides a summary of common malware functions and delivery mechanisms, taditional malware defenses, and recent malware trends. Sections 1.2–1.3 introduce behavioral malware analysis and system call tracing. Section 1.4 describes the motivations for and challenges in performing behavioral malware analysis online, and Section 1.5 emphasizes the importance of empirical evaluation. Section 1.6 summarizes the research contributions of this thesis.

1.1 Malware

Malware are traditionally characterized by their malicious functions, delivery mechanisms, and authorship or heritage. The following categories describe common malicious functions.

Backdoors

Backdoors provide clandestine remote access to an infected host, typically bypassing normal authentication and security mechanisms.

Bots

Bots enable remote control of a host. The infected hosts are typically part of a network of infected hosts, referred to as a *botnet*. Bots are used for performing distributed operations, such as sending spam, mining crypto-currency, or performing DDoS attacks [1, 2].

Downloaders and droppers

Downloaders and droppers install additional malware on infected hosts. Downloaders download additional malware from remote servers, while droppers are bundled with additional malware.

Ransomware

Ransomware restrict access to a host, by disabling or hiding normal functions, or by encrypting or hiding data. Ransomware are typically used to extort money from affected users [3].

Rootkits

Rootkits covertly provide persistent and privileged access to a host. Rootkits typically hide processes, hide files, and interfere with security software to evade detection.

Scareware

Scareware use fear to sell a typically useless product. For example, rogue antivirus (AV) software erroneously report malware infections and are used to sell the service removing the reported infections [4].

Spyware

Spyware steal private user information, such as financial data, browsing habits, license keys, and login credentials. Spyware typically transmit the stolen information to a remote server.

Malware are also characterized by their delivery mechanisms, the methods by which they spread to and infect new hosts. The following malware categories describe common delivery mechanisms [5].

Exploits

Exploits take advantage of software vulnerabilities to enable privileged execution of malware.

Logic bombs

Logic bombs are malicious functions embedded in otherwise benign software.

Trojan horses

Trojan horses, also known as Trojans, are malware disguised as legitimate software, used to trick users into unknowingly installing malware.

Viruses

Viruses are self-replicating malware that insert code into other programs, files, or locations on an infected host.

Worms

Worms are self-replicating malware that spread to other hosts, typically over a computer network.

To maximize their effectiveness and evade defenses, malware samples typically include multiple malicious functions and use multiple delivery mechanisms. As such, malware samples typically exhibit characteristics of multiple categories.

1.1.1 Malware naming

No single, universal malware taxonomy or naming scheme exists. Instead, multiple naming schemes have been developed by major security companies to identify malware samples. Such naming schemes typically include a *malware category* label, which indicates the primary delivery mechanism and/or malicious function of a malware sample, and a *malware family* label, which indicates specific shared function, authorship, or heritage. Examples of malware families include Zeus/Zbot, Flame, Stuxnet, Duqu, MyDoom, and Sality. Often, naming systems include a *sub-family* or *variant* label, indicated by an alphanumeric identifier. Naming systems may also specify the target platform of a malware sample or the language in which it is written.

The existence of malware samples that use multiple delivery mechanisms and perform multiple malicious functions complicate the malware naming process. Although the functions and delivery mechanisms listed in Section 1.1 are not mutually exclusive, malware naming schemes typically indicate only the primary function or delivery mechanism of a sample. Existing malware naming schemes suffer from two major drawbacks: *incompleteness* and *inconsistency* [6]. The naming schemes are incomplete because there are known malware samples missing from each scheme. The major challenges to creating a complete naming schemes are the number of existing malware samples and frequency at which new malware samples are discovered. More than 50 million new malware

Table 1.1: Names assigned to two malware samples by five antivirus (AV) vendors

vendor	sample A	sample B
Microsoft	PWS:Win32/Zbot	PWS:Win32/Zbot.AEO
Kaspersky	Trojan-Spy.Win32.Zbot.tivt	Trojan-Dropper.Win32.Injector.crie
ESET	Win32/Injector.BGXN	*None*
McAfee Labs	*None*	Generic Dropper!1jk
Dr. Web	Trojan.PWS.Panda.655	Trojan.PWS.Panda.655

samples were discovered in the fourth quarter of 2014 alone, an average of 6 new malware samples every second. McAfee Labs projects to have more than 500 million total samples in their collection by the end of 2015 [7].

Existing malware naming systems are inconsistent because different naming schemes do not place malware samples consistently in the same category or family. One of the major challenges to achieving consistent malware naming is the overlap in malware functions and delivery mechanisms. Furthermore, malware samples are often composed of code derived from multiple different sources, complicating the problem of establishing malware authorship or heritage. Finally, malware samples are typically designed to complicate the types of analyses used to reveal their functions, delivery mechanisms, authorship, and heritage. Techniques used by malware authors to complicate such analyses include obfuscation, packing, encryption, anti-debugging, anti-disassembly, and other anti-analysis techniques [8].

The inconsistency and incompleteness of malware naming schemes are demonstrated in Table 1.1 The table shows the names assigned to two malware samples by five different AV vendors, Microsoft[1], Kaspersky[2], ESET[3], McAfee[4], and Dr. Web[5]. The names are presented in the formats provided by the vendor. The incompleteness of the naming schemes is demonstrated by the missing name for sample A from McAfee and the missing name for sample B from ESET. The inconsistency of the naming schemes is demonstrated through a comparison of the labels assigned by each vendor. For example, Microsoft identifies both samples as distinct variants belonging to the password stealer

[1]Microsoft Malware Protection Center (MMPC), `http://www.microsoft.com/security/portal/mmpc/`
[2]Kaspersky Lab, `www.kaspersky.com`
[3]ESET NOD32, `www.eset.com`
[4]McAfee Labs, `http://www.mcafee.com`
[5]Dr. Web, `www.drweb.com`

(PWS) category and Zeus/Zbot family. Dr. Web similarly identifies the samples as being members of the PWS category and from the same family, but does not differentiate between the two samples. Kaspersky identifies the samples as belonging to two distinct categories, Trojan-Spy and Trojan-Dropper, and two distinct families, Zbot/Zeus and Injector.

1.1.2 Malware delivery mechanisms

Malware primarily infect hosts through the exploitation of security vulnerabilities, through the exploitation of lax security practices, and through social engineering. Security vulnerabilities are used by malware authors to enable privileged malware execution. These include input validation, code injection, buffer overflow, privilege escalation, and cross-site scripting vulnerabilities [9]. While many vulnerabilities are discovered and patched by the authors of affected software, others are discovered and exploited by an attacker before the vulnerabilities are known. Such vulnerabilities, known as *zero-day vulnerabilities*, are particularly worrisome for commonly used software, as they enable malware authors to successfully infect many vulnerable hosts. Furthermore, malware exploiting zero day vulnerabilities can spread largely unchecked until the exploited vulnerabilities are discovered and patched.

Malware targeting hosts with lax security typically seek easy targets, such as hosts with weak passwords. For example, some malware scan computer networks for hosts running common network services. The malware attempt to access the discovered services by performing dictionary attacks with common username and password combinations.

Social engineering is the use of disguises, ploys, or tricks to convince a user to unknowingly assist an attacker in infecting a host [10]. One common social engineering tactic is the use of Trojan horses to trick users into downloading and installing malware. For example, the Zeus/Zbot malware family has been used in targeted email campaigns, sent to potential victims as electronic greeting cards from the White House, Federal Trade Commission complaints, and shipping invoices.

Of these three delivery mechanisms, the exploitation of security vulnerabilities is comparatively rare. This is largely due to the engineering effort required to identify exploitable security vulnerabilities. When vulnerabilities are discovered, they are typically sold on the black market to be used in cyber-attacks [11]. Furthermore, exploits are only useful as long as the targeted vulnerabilities remain open [12]. Conversely, social engineering requires only a sufficiently convincing ploy to trick targeted users, and the discovery of computer hosts with lax security is easily automated.

1.1.3 Malware motivations

Malware authors are motivated by a number of factors, including politics, ideologies, and monetary gain. Politically and ideologically motivated malware are increasingly being used as parts of sophisticated, targeted attacks known as advanced persistent threats (APTs) [11]. APTs typically use multiple delivery mechanisms to gain access to a target's computer systems. Examples of APTs include Stuxnet, used for sabotage at an Iranian nuclear facility [13], and Operation Aurora, used for corporate espionage [14].

Malware are also particularly useful as profit source. Some malware, such as Zeus/Zbot, are sold in the form of crimeware kits used for creating customized malware samples [15]. Malware authors also sell access to botnets, which are typically used to provide phishing and spam services, to mine crypto-currency, and to perform DDoS attacks [16]. Spyware are used to steal sensitive data, such as user credentials and financial data, which are sold in the black market [17]. Ransomware and scareware are used to extort money from the users of infected hosts [4].

1.1.4 Malware defenses

Traditional defenses against malware include the adoption of secure computing practices, the design of more secure systems, and the use of security-related software tools. Secure computing practices include

- keeping software up-to-date to mitigate the chance of known vulnerabilities being exploited [4];
- safeguarding private information against casual reconnaissance; and
- adopting security policies to reduce the number of possible attack vectors on a host.

Teaching secure computing practices to end-users is an especially important step in preventing intrusions enabled by lax security practices, slowing the proliferation of Trojan horses, reducing the success of phishing attempts, and thwarting other attempts at social engineering [18, 19].

There has also been a considerable effort to design computing systems with security in mind, a field known as *trusted computing* [20]. Techniques used in trusted computing include protecting sensitive information at the hardware-level and signing code so users can verify whether it comes from a trusted source. Recent malware trends indicate that malware authors have begun digitally signing their malware to avoid detection by such mechanisms. McAfee reports nearly 16 million distinct signed malware samples have been discovered through the beginning of 2015 [7].

Software tools used to protect against malware infections include intrusion detection systems (IDSs) and AV software. IDSs are used to detect malware and other malicious intrusions. Host IDSs detect intrusions at the host-level, by monitoring operating system (OS) interaction, file access, and audit data [21, 22]. Network intrusion detection systems (NIDSs) detect intrusions at the network level, by monitoring network traffic. IDSs can be signature-based, detecting intrusions based on patterns of known attacks [23], or anomaly-based, detecting intrusions by identifying deviations from normal behavior [24, 25].

AV software typically identify files as either benign or malicious by comparing them against a database of byte signatures of known malware [5]. The strengths of AV software are that they provide a low false positive rate (FPR) and have easily provable and quantifiable performance against known malware samples. However, the weakness of signature-based AV software is that they are reactive, typically requiring new malware to be discovered and analyzed prior to detection. Furthermore, AV software to fail to detect new malware variants that have been modified just enough to not match any existing signatures. In response, AV software incorporate heuristic techniques to identify common types of malware, and signatures are designed to detect as many variants of a malware sample as possible. Techniques for streamlining malware analysis and reducing the number of required signatures are of particular interest to AV vendors [7, 26].

1.1.5 Recent malware trends

To evade detection by AV software, malware authors generate new malware variants by reordering, encrypting, recompiling, compressing, padding, or otherwise changing malware samples without altering their function [8]. Such changes can be applied automatically using specialized software, such as encrypters and packers. Certain types of malware modify themselves automatically as they propagate. Polymorphic viruses mutate by re-encrypting critical parts of themselves, while metamorphic viruses mutate by changing their own code structure as they propagate [27, 28]. As malware authors have become more savvy at creating new malware variants, the burden on AV vendors to create signatures to detect all of the variants has increased. The time it takes for AV software to detect a new malware sample varies from a few days to many months [29].

New malware variants can be released over the course of many months or years. For example, the Zeus/Zbot malware family was first discovered in 2007. In 2010, more than 90,000 new, distinct variants were discovered [11]. More than 8 years after its introduction, new variants of Zeus/Zbot continue to be discovered. Another popular technique for releasing malware variants is the *variant flood* attack. In a variant flood attack, tens of thousands of variants are released in rapid succession. The goal of such an attack is to overwhelm AV vendors, who struggle to quickly analyze and generate signatures for all of the variants [30].

Computing infrastructure continues to grow in both size and complexity, as illustrated by recent trends such as the emergence of cloud computing and ultra large scale (ULS) systems [31]. This increase in complexity introduces new security vulnerabilities, while simultaneously complicating the detection of malware-based cyber-attacks. The combined effects of the increasing prevalence and sophistication of malware, the vulnerability of traditional AV software to malware variants, and the increasing complexity of computing infrastructure, motivate the development of techniques to detect the malware samples that evade traditional defenses. Among the most promising techniques are those from behavioral malware analysis, described in the following section.

1.2 Behavioral malware analysis

Among the techniques proposed to address the weaknesses of traditional malware detection methods are those based on *dynamic analysis* or *behavioral analysis*. In behavioral analysis, observed characteristics of executing software are used to detect the presence of malware. Behavioral characteristics are a desirable source of information, since they are typically unaffected by the obfuscations used to evade traditional AV software. One of the major challenges in designing a behavioral malware detector is to determine a set of behavioral characteristics that

- can be used to differentiate between malware and benign software;
- can be used to differentiate among different families and categories of malware;
- are robust to differences in benign functions, configurations, workloads, and usage patterns;
- are difficult for malware authors to obfuscate; and
- are easily observable in the target environment.

The first two criteria focus on the discriminability afforded by the characteristics, *i.e.*, whether they contain enough information to detect and classify malware. Not only must the characteristics provide dicrimination, they must must also afford high detection rates at very low false positive rates (FPRs) [32]. The FPR of a malware detector is especially important because of the relatively low incidence of malware infections when compared to the number of events that a malware detector observes. How the FPR of a detector is characterized depends on the type of detector. For AV software, the FPR might be characterized as the fraction of all benign files that are incorrectly identified as malware [33]. For an IDS, the FPR might be characterized as the fraction of observed events misclassified as malicious. The FPR of a detector can also be characterized as the average time between false positives.

The third criterion focuses on the robustness of the chosen characteristics. To be applicable to a wide variety use-cases, the characteristics should be unaffected by differences among benign systems, such as host and software configurations. The chosen characteristics should also be unaffected by benign changes in workload or usage patterns.

The fourth criterion focuses on the robustness of the characteristics in terms of the estimated level of effort required for obfuscation. In the case of static signature-based detection, the required level of effort is low. Malware authors employ an automated process of generating new variants and comparing them against the latest available AV signatures. Variants that are undetected are deemed suitable for deployment. If appropriate behavioral characteristics are chosen for detection, there is no longer a simple, well-defined obfuscation process to evade detection. The challenge for malware authors changes from the relatively easy changing of the structure of a static file to evade detection, to the comparatively more difficult altering of the behavioral characteristics of a program while maintaining its malicious function. The most successful type of attack against a behavioral malware detector would likely be a *mimicry attack*. In a mimicry attack, malware samples are designed to behave more like benign software to evade detection [34]. *Shadow attacks* are another possible type of attack, wherein malicious behaviors are divided among multiple processes [35]. Such attacks are successful against process-level system call signature detectors when crafted to obfuscate specific detection signatures. The selected behavioral characteristics must be difficult for malware authors to obfuscate while maintaining malicious functionality.

The final criterion concerns the observability of the chosen characteristics on the target platform. First, observation of the chosen characteristics must be feasible on the target platform to even be considered. For example, the entire control flow graph of software program may be useful for characterizing software function and detecting malware. However, such information is difficult to reliably extract, limiting its practical applicability [36]. Second, the desired characteristics may be observable, but the costs associated with extracting the required information may be prohibitive. For example, an accounting of all of information flow through a host may be useful for identifying privacy-breaching malware. However, the overhead introduced by monitoring the information flow is very high, precluding its use in complex production systems [37]. The problem of selecting the appropriate characteristics to use for behavioral malware detection is addressed in Chapter 4.

1.3 System call tracing

One particularly promising source of behavioral characteristics for malware detection are system call traces. A system call trace is a detailed accounting of the system calls occurring on a host. System calls are a mechanism used by software programs to request OS services. In the Microsoft Windows family of OSs, the set of all 465 system calls (although not officially documented) can be partitioned into the following categories [38],

- atoms (for string manipulation),
- boot configuration,
- debugging,
- device driver control,
- environment settings,
- error handling,
- files and general input/output (I/O),
- jobs,
- local procedure calls (LPCs),
- memory management,
- miscellaneous,
- object management,
- plug and play,
- power management,
- process and thread management,
- processor information,
- registry access,
- security functions,
- synchronization, and
- timers.

System call traces are of interest for malware detection because they provide information about how executing software interact with a host OS. In particular, system call-based malware detection works under the assumption that the function of a program can be coarsely determined by observing its system call trace [39]. This assumption applies to malware because their malicious functions (described in Section 1.1) require the use of OS services. A major challenge in system-call based malware detection is the determination of feature set to extract from system call traces that satisfies the criteria described in Section 1.2.

The relative ease of obtaining system call traces also makes them a desirable source of information for malware detection. Traces can be obtained using the `strace` program or `auditd` framework in Linux, or the Event Tracing for Windows (ETW) framework in Windows. A system call trace is typically a log of two types of events, system call enters and system call exits. System call enter events typically include information about the processor from which a system call originates and the data passed as input to the call. System call exit events typically include the data returned by a system call. The rate of system calls on a host varies based on processor speed, number of

```
NtQueryPerformanceCounter
NtProtectVirtualMemory
NtProtectVirtualMemory
NtQueryInformationProcess
NtProtectVirtualMemory
NtQueryInformationProcess
NtQueryInformationProcess
NtQueryInformationProcess
NtQueryInformationProcess
NtQuerySystemInformation
NtQuerySystemInformation
NtAllocateVirtualMemory
NtFreeVirtualMemory
NtAllocateVirtualMemory
NtQuerySystemInformation
NtFreeVirtualMemory
NtOpenDirectoryObject
NtOpenDirectoryObject
NtOpenSymbolicLinkObject
NtQuerySymbolicLinkObject
NtClose
NtAllocateVirtualMemory
NtQueryVirtualMemory
NtOpenFile
NtQueryVolumeInformationFile
NtQueryAttributesFile
NtOpenFile
NtCreateSection
NtMapViewOfSection
NtQuerySection
```

Figure 1.1: Example system call trace showing the first 30 calls made by a web browser process

processor cores, and type and intensity of computational load. On the hosts used in this study, the rate at which system calls are made ranged from hundreds of system calls per second to hundreds of thousands of system calls per second.

Another advantage to using system call tracing for malware detection is that system call traces can be collected in real-time on production hosts. By logging only a subset of the data – *e.g.,* by ignoring input arguments, ignoring returned data, sampling the data, or aggregating the data – the memory, processing, and storage overhead required for system call tracing can be tuned to enable live data collection and processing.

Figure 1.1 shows an example system call trace in the format used for this study. It is a process-level trace of a web browser, beginning when the web browser process was created. It shows the system calls made by the web browser process in the order in which the corresponding system call enter events were observed. Only the first thirty system calls of the trace are presented for brevity.

This system call format is advantageous because the sequence of system calls depicted in Figure 1.1 can be concisely encoded and stored for later analysis. Such system call traces can be collected at multiple different levels, including the host, application, process, and thread levels.

1.4 Detecting and classifying unknown malware samples

Section 1.1.4 identified two types of malware against which traditional defenses struggle, *i.e.*, new malware samples and new variants of existing malware samples. Such malware are typically free to perform their malicious tasks until they are discovered. Even if the security vulnerabilities or lapses that led to the infection are corrected, or the processes where the infection originated are terminated, the malware samples typically have already gained privileged access to the host and will remain active. Such samples are discovered by users or administrators who observe abnormal behavior that can be attributed to the malware infection. The discovery of such malware infections is not straightforward, as many malware samples are designed to hide their presence.

Historically, malware infections were identifiable through obvious indicators, such as degradation of system performance, excessive network traffic, excessive hard drive access, or frequent crashes. As malware become more sophisticated and focused on stealth, characteristics traditionally associated with malware infection have become less pronounced. For example, three malware families – Stuxnet, Flame, and Duqu – ran undetected in production environments for over a year before they were discovered [13, 40–42]. Such malware samples serve as motivating examples for this study.

The goal of this study is to establish a set of techniques for identifying malicious processes, especially those resulting from the execution of new malware samples and malware variants that evade traditional defenses. Furthermore, the goal is to detect such infections as quickly as possible, to limit the adverse effects of the malware. The three primary components of this thesis are the study of the problems of feature extraction, malware detection, and malware classification.

Feature extraction

The objective of feature extraction is to identify a set of features that can be extracted from system call traces and that satisfy the criteria described in Section 1.2. The extraction of the selected features must be simple enough that it can be performed in real-time on production hosts. This study uses frequency information about localized patterns of system calls as its primary feature set. The data are extracted using information retrieval techniques inspired by document classification. This study also seeks to identify the smallest set of features to use for detection. Using the smallest possible set of features serves to reduce the memory and computational overhead of the system and to avoid the adverse effects of detector and classifier overfitting.

Malware detection

The objective of malware detection is to establish a set of techniques for detecting malware execution in real-time on production hosts. This work compares the empirical detection performance of signature-based and statistical anomaly-based detection techniques that are 'lightweight' enough to be used in production environments. The techniques are evaluated against system call traces collected from production environments and from the execution of recently discovered malware samples. Emphasis is placed on characterizing detector performance at very low FPRs and on performing detection as quickly as possible using sequential detection techniques.

Malware classification

The goal of the malware classification is to establish a set of techniques to classify new malware samples based on their behavioral similarity to known malware. Classification results provide insight into the damage caused by detected malware samples and possible paths for mitigation. The focus of this study is on classification techniques that use the same behavioral features as the detector. The techniques studied in this thesis can be applied to classify malware samples immediately as they are detected, without requiring any additional data collection.

1.5 The importance of empirical evaluation

To be useful in production environments, a malware detection system must exhibit a high detection rate at an acceptably low FPR [32]. Furthermore, detection performance must be quantified in environments representative of those in which the system is likely to be deployed, or preferably quantified using data collected from the target environments [43].

The requirement to evaluate proposed detection and classification results in representative environments precludes the use of specialized sandbox environments. Sandbox environments provide detailed information about software execution, including filesystem, registry, process, and system call activity [44]. However, sandbox environments typically limit execution paths and provide a limited view of software function [45]. Thus, data collected from such environments are typically not representative of data collected from production environments.

A major component of this thesis is the empirical evaluation and comparison of the described feature extraction, malware detection, and malware classification techniques. This study used a custom host-agent to collect system call traces from production hosts running 32-bit and 64-bit versions of Windows 7 and 8, and Windows Server 2008 (R2) and 2012 (R2). A custom malware testbed was used to inject recently discovered malware samples onto live hosts. The empirical evaluation considered more than 55,000 malware samples first discovered between January 2012 and March 2015 and 4 million system call traces collected from 43 hosts. In total, more than 1,000 host-days of data collected between October 2014 and March 2015 were used for the experimental evaluation of the described techniques.

1.6 Research contributions and thesis outline

The preceding sections introduced three key challenges in behavioral malware detection and classification, including

- feature extraction from system call traces,
- rapid detection of malicious processes at a low FPR, and
- classification of detected malware using only the features available at detection time.

The remaining chapters of this thesis address these three challenges, describing a system for detecting the execution of malicious processes on production computer hosts. The system, depicted in the block diagram in Figure 1.2, uses a custom host-agent to track the system call traces of every process executing on a host. From the traces, it extracts numerical feature data that are used by the detector to provide binary decisions indicating whether a process is benign or malicious. If

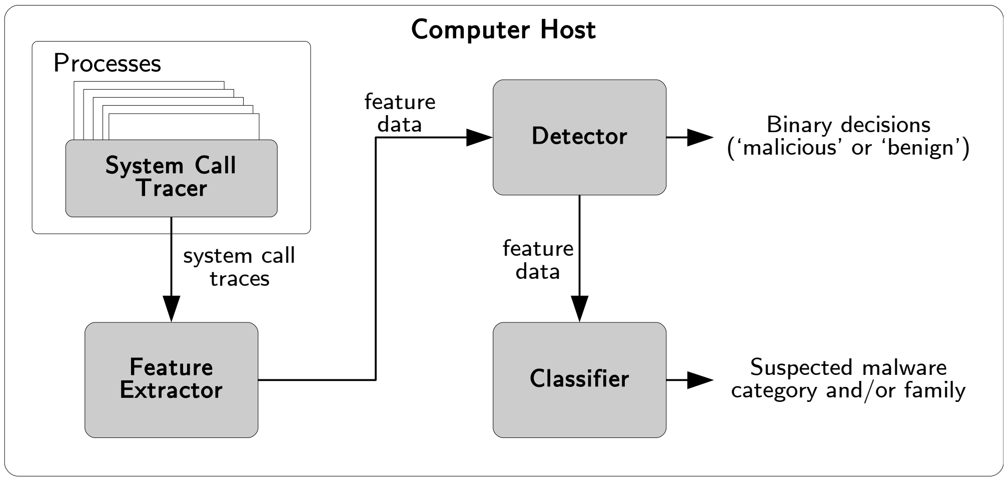

Figure 1.2: Block diagram of malware detection and classification system

the detector identifies suspected malware, the classifier determines the most likely category and/or family to which the malware belong. The specific configuration of the system is determined through experimental evaluation of multiple feature extraction, detection, and classification strategies.

The primary differences between this work and the related literature are presented in Chapter 2. Chapter 3 describes the design and deployment of the data collection host-agent. Chapter 4 provides a detailed overview of the feature extraction process. Chapter 5 examines the performance afforded by four detection techniques. Sequential detection techniques are also explored to provide a continuous monitoring scheme for malware detection and to provide more rapid detection results than are afforded by fixed sample-size tests. Through experimental evaluation, the feature extraction strategies, detection algorithms, and sequential detection strategies that provide the best detection performance are identified. The selected strategies are used in a case study to test the detection system's performance against an independent data set collected from public computer laboratories. Chapter 6 describes five classification strategies for identifying the category and family of detected malware samples based on their behavioral similarity to known malware. It compares the accuracy of the classifiers using 27 ground truth labeling schemes derived from AV labels. Chapter 7 concludes with a summary and discussion of the experimental results, the resulting design of a behavioral malware detection and classification system, and the implications of this study.

Chapter 2: Related Work

This chapter presents a review of the related work in malware detection and classification, with a focus on behavioral analysis techniques developed to address the shortcomings of traditional defenses. Additional related work can be found in surveys by Idika and Mathur [46], and by Egele, Scholte, Kirda, and Kruegel [44]. A detailed discussion and comparison of commercial antivirus (AV) software can be found in a survey by Sukwong, Kim, and Hoe [29].

2.1 Malware detection

The related work in malware detection seeks to identify techniques that are more robust than existing byte signature and heuristic techniques popular among AV software. The proposed techniques include static analysis techniques, in which features are extracted from static files, and dynamic analysis techniques, in which features are extracted from executing software.

2.1.1 Static analysis techniques

Static analysis techniques use features of static files to identify potential malware. Early work in static analysis focused on byte code analysis, using data mining and machine learning instead of traditional signature generation techniques [47]. Entropy-based byte code analysis techniques have also been proposed to detect packed, encrypted, or embedded malware [48]. Other useful static features include the strings, file headers, imported libraries, and calls to external application programming interface (API) functions present in a static executable file [49–51]. Even the approximate authorship dates of the functions composing a program are useful features for malware detection [52].

Control flow information is also useful for malware detection. For example, formal semantics can be used to describe common malicious behaviors in terms of their control flow, and the resulting models can be used to detect semantically equivalent code [53, 54]. Other approaches use control flow information to detect computer worms [55], web browser spyware [56], and metamorphic malware [57–59]. However, techniques for detecting semantically equivalent code have demonstrated to be effective only against specific obfuscations [60]. Furthermore, static analysis techniques relying on relatively high-level features – such as control flow graphs – are limited by the difficulty of extracting the required information. Such techniques typically require malware first be unpacked and disassembled [61]. Unfortunately, malware are often encrypted, compressed, or otherwise designed specifically to complicate such analyses [36].

2.1.2 Dynamic analysis techniques

Dynamic analysis techniques have been widely studied as a means of addressing the shortcomings of static analysis techniques. One such technique, taint analysis, identifies malicious behaviors by tracking information flow [45, 62–66]. Taint analysis is particularly useful for identifying privacy-breaching malware. It works by monitoring access to sensitive data and identifying potentially malicious use of such data. However, its computational overhead, high false positive rate (FPR), and vulnerability to relatively simple countermeasures limit its practical applicability [37, 67].

Other dynamic analysis techniques use data collected from hardware performance counters, which provide information about processor usage, such as cache misses and branch predictions [68, 69]. Software performance monitors, such as those that report CPU, memory, network, and application-specific resource usage information, are also useful features for malware detection [70–72]. Sensors specific to the Java Virtual Machine have also been used to detect the execution of malicious Java code [73]. Other techniques mine audit data [74] or monitor registry and filesystem operations [75] for indicators of malware infections.

2.1.3 System call analysis

System call traces (described in Section 1.3) have been a popular source of information for dynamic malware detection. The use of system call traces for malware detection originated in intrusion detection, where abnormal system call sequences were flagged as evidence of potential intrusions [22, 39]. System call-based malware detection has been approached using both signature-based and statistical techniques. Signature-based techniques seek to identify system call patterns unique to malware samples, while statistical techniques seek models to describe benign and malicious behaviors.

Table 2.1: Summary of related work in system call-based malware detection

Strategy	Features	Detection	Evaluation
Forrest *et al.* [39]	call sequences	signatures	intrusion detection
Hofmeyr *et al.* [22]	call sequences	signatures	intrusion detection
Warrender *et al.* [76]	call sequences, call frequency	HMM, rule learner	intrusion detection
Liao and Vemuri [77]	call frequency	nearest neighbor	intrusion detection
Kang *et al.* [78]	call frequency	SVM, LR, naïve Bayes, decision tree, rule learner	intrusion detection
Xin and Xu [79]	call sequences	Markov models, temporal difference learning	intrusion detection
Burguera *et al.* [80]	call frequency	clustering	3 benign, 5 malicious (3 self-written)
Martignoni *et al.* [65]	activities	graph matching	11 benign, 7 malicious
Tokhtabayev *et al.* [81]	activities	signatures	210 benign, 31 malicious
Mehdi *et al.* [82]	call sequences	rule learner, SVM, decision tree	72 benign, 72 malicious
Kirda *et al.* [56]	activities	signatures	18 benign, 33 malicious
Kolbitsch *et al.* [83]	data flow	graph matching	5 benign, 563 malicious (6 families)
Pfoh *et al.* [84]	call sequences	SVM	285 benign, 1943 malicious
Xiao and Stibor [85]	call sequences	latent Dirichlet allocation	168 benign, 2880 malicious
Canali *et al.* [43]	{sequences, tuples, bags} of {calls, activities, arguments}	signatures	363k benign, 7k malicious
Lanzi *et al.* [75]	call sequences, activities	signatures	362k benign (242 applications), 10k malicious

A summary of the related work in system call analysis for malware detection is provided in Table 2.1. The table presents the primary feature sets and detection algorithms used in each study, along with a concise summary of the data used for experimental evaluation. The majority of the works use system call sequences and frequencies as feature sets. Others argue system calls to be too low-level, instead considering high-level activities extracted from system call traces. Such activities included filesystem and registry changes, library interactions, and process creation and deletion events. For detection, signature-based approaches are popular due to their white-box models and low computational complexity during detection. Other approaches include the use of machine learning and graph matching algorithms. Graph matching algorithms are used to compare the control flow or data flow of a program to models of known benign and malicious behaviors. Such techniques are intended to address the shortcomings of signature-based techniques, namely that they do not generalize well to detecting unknown malware.

The approaches presented in Table 2.1 are partitioned into three groups, indicated by the horizontal lines in the table. The studies in the first group [22, 39, 76–79] use system call trace data for intrusion detection. These studies focus primarily on system call sequences and system call frequencies as feature sets, using machine learning algorithms for detection. The techniques are evaluated against system call traces of a small collection of Unix program, obtained under normal operating conditions and during suspected intrusions.

The studies in the second group [56, 65, 80–85] use system call trace data for malware detection. Like the intrusion detection work in the first group, a subset of these techniques use system call sequences and frequencies for detection. Other feature sets derived from system call traces are also considered, including models of malicious and benign activities [56], and data flow among system calls [83]. The table indicates the quantity of benign and malicious software samples used for the experimental evaluation of these techniques.

The studies in the third group [43, 75] also use system call trade data for malware detection. The two studies in this group are set apart by their experimental evaluation. Whereas the studies in the second group consider only a few malware samples or benign programs, the studies in the third group consider thousands of malware samples and hundreds of thousands of traces of benign programs from multiple hosts. Lanzi *et al.* [75] use system call signatures for detection, concluding that such an approach is inadequate for malware detection. Instead, they present a system-centric approach that uses models of filesystem and registry activities for detection. Canali *et al.* [43] take a signature-based approach, using system calls, their arguments, and derived activities for malware detection. The authors explore the use of contiguous sequences (sequences), non-contiguous sequences (tuples), and unordered occurrences (bags) of calls and activities for detection. The results of this work indicate that there is no closed form expression to describe the effects of model selectivity or specificity on detection accuracy, underscoring the importance of experimental evaluation.

2.2 Malware classification

Malware classification is the process of determining to which class a particular malware sample belongs. Classes can be malware categories, families, subfamilies, or any other malware groupings. Automatic classification can help guide malware analysis by providing information about the origin, authorship, or function of a malware sample. In live environments, classification is useful for determining appropriate mitigation strategies and guiding post-mortem analysis. A large portion of related work in malware classification focuses primarily on systems designed for off-line analysis. Such work is motivated in part by the need to expedite the analysis of and signature creation for newly discovered malware samples. The related work in malware classification includes both static and dynamic analysis techniques.

2.2.1 Static analysis techniques

The feature sets used for static classification mirror those used in static detection, including

- strings, byte sequences, and program structure [47, 86, 87];
- API imports, calls, and call sequences [88]; and
- control flow information [89–91].

The classification algorithms used for static malware classification are primarily machine learning techniques. Commonly used classification algorithms include

- naïve Bayes, decision trees, and support vector machines (SVMs) [47, 87];
- image classification techniques [86];
- hierarchical clustering analysis [88]; and
- graph matching and clustering algorithms [89–91].

2.2.2 Dynamic analysis techniques

Dynamic classification techniques typically require malware samples be executed in a specialized analysis environment. Such environments, including Anubis [92] and cwsandbox [93], provide reports that include information about loaded libraries and other resources; file and registry activities (files and keys created, read, written, and deleted); memory activities (regions read and written); and system call traces. Nearest neighbor techniques have been used to classify malware based on such reports [93–95], with one study focusing on the problem of determining the best distance measure to assess similarity [96]. Alternate machine learning algorithms have also been explored, including naïve Bayes, SVM, decision trees, and neural networks [95]. Hierarchical clustering analysis based on file, registry, and process activity have also demonstrated promise [6]. Techniques have been described for classifying specific subsets of malware, including malware that communicate over a network [97, 98] and certain rootkits that use API hooks [99].

As with malware detection, a subset of the classification techniques also focus primarily on system call analysis. For example, system call traces extracted using cwsandbox and nearest neighbor techniques have demonstrated success in classifying malware [100, 101]. Other techniques have used activities derived from system call traces and clustering algorithms to perform classification [102].

Two recent studies combined static and dynamic analysis techniques for malware classification. Neugschwandtner *et al.* [26] use static analysis based on a variety of features – including byte sequences, program structure, program headers, and AV labels – to cluster malware according to predicted behaviors. Then, they use dynamic analysis to analyze malware behaviors and generate mitigation procedures. Anderson *et al.* [103] use SVMs and multiple static and dynamic data sources – including byte sequences, op-codes, control flow graphs, instruction traces, and system call sequences – for classification. These two studies leverage the strengths of static analysis (ease of feature extraction) and dynamic analysis (richness of datasets) to perform malware classification.

2.3 Novelty of this thesis

This chapter explored the related work in malware analysis, describing feature sets and algorithms used for malware detection and classification. Despite their differences, these techniques all exhibited promising empirical accuracy against their respective datasets. This thesis seeks to address two perceived shortcomings of the related work. First, the related work contains conflicting claims regarding the effectiveness of feature sets, detection, and classification techniques. Second, the related work lacks a study of detector performance at very low FPRs. This thesis also contributes the design of an online detection system for malicious processes, a study of sequential detection techniques for rapid malware detection, an extensive experimental evaluation, and the design of an online classification system for malicious processes.

2.3.1 Addressing conflicting claims

Published results in malware detection contain conflicting claims. For example, signatures of system call sequences have been declared both effective [39, 101] and ineffective [6, 75] for detecting malware. Similarly, system call frequencies [78, 80] have been found effective for malware detection, while other works advocate for more complex feature sets [79, 85]. These conflicting claims are at least partially rooted in the use of different datasets for evaluation. Accordingly, this study compares the usefulness of different types of features, including system call frequencies, ordered sequences of system calls, and unordered tuples of system calls. This study also compares signature-based techniques and statistical machine learning techniques for malware detection and classification. The techniques are evaluated against a single set of system call traces collected from live computer hosts and recent malware samples to ensure fair comparisons.

2.3.2 Detector performance at low FPRs

For malware detectors to be practically useful, they must demonstrate high detection rates at very low FPRs [32]. For the majority of the related work, few claims regarding detection accuracy at very low FPRs or the general applicatbility of the proposed technuques can be made. This is due to the relatively small sample sizes considered for experimental evaluation. The two exceptional cases in the related work were from Canali *et al.* [43] and Lanzi *et al.* [75], who used in excess of 100,000 benign traces in their evaluation. Canali *et al.* [43] provided a characterization of detection rate at a FPR of 10^{-2}, and characterized certain models at lower FPRs ($\sim 10^{-3}$). The experimental evaluation presented in this thesis focuses on the detection accuracy achieved at a FPR of 10^{-5}.

2.3.3 Online detection

The goal of this study is to detect and classify the malware that evade traditional defenses and execute on production hosts. Particularly, this study focuses on techniques that are 'lightweight' enough to be deployed online in production environments. In this way, this study is similar to related work in intrusion detection (see Section 2.1.3), but with a singular focus on malware detection. This work intends to identify techniques for malware detection that will be useful in identifying other types of intrusions as well.

2.3.4 Sequential detection

A major concern in online malware detection is detecting malware quickly enough to mitigate their damaging effects. To address this concern, this thesis studies the application of sequential detection techniques to malware detection. A subset of the related work applies sequential detection techniques to network data for the detection of worms and other network-based attacks [25, 104–106] and for the detection of multi-stage cyberattacks [79]. In contrast, this work focuses on the application of sequential detection techniques at the process level. Here, processes undergo continuous monitoring, wherein successive detector outputs are combined to provide accurate detection results while minimizing the delay between infection and detection.

2.3.5 Experimental evaluation

In the related work, the source of the system call traces used for experimental evaluation varied. A common theme was the use of malware traces collected from specialized analysis environments. Such sandbox environments limit execution paths, thereby offering a limited view of software functionality [45]. Sandbox environments typically track only a fraction of the system calls available in recent versions of Microsoft Windows. Another common theme was the use of benign process traces collected using API hooking and system service descriptor table (SSDT) hooking in Microsoft Windows XP [43, 75, 84, 85].

In contrast, this thesis focuses exclusively on system call traces obtained from live production hosts. This includes the execution of recently discovered malware samples on live hosts configured like those found in production environments. Executing malware on live hosts enables the malware to behave more closely to how they would in a real-world environment. Furthermore, doing so reduces the risk of biasing results by using traces collected from different environments. This thesis studies the problem of malware detection in recent versions of Windows, including Windows 7, 8, Server 2008 (R2), and Server 2012 (R2). The techniques used for system call tracing in this study are more robust than the techniques of API hooking and SSDT hooking used in the related work.

2.3.6 Online classification

The related work in behavioral malware detection focused primarily on classification of malware based on rich feature sets extracted using specialized environments. This study seeks to use a more limited dataset, namely only those system call features used by the described malware detector. The classifier uses the information already available when malware samples are detected on production hosts to provide immediate classification results. The motivation for online classification is to provide information for mitigation and postmortem analysis. Prior information about similar threats can be used to suggest possible mitigations, and classification results can be used to guide the use of more rigorous static and dynamic analysis tools. Thus, the goal of this study is not to outperform more rigorous techniques, but rather to achieve comparable classification accuracy in production environments with a more limited feature set.

Chapter 3: Experimental Setup

Experimental evaluation is essential for determining the effectiveness of malware detection and classification systems [32, 43]. To evaluate the effectiveness of the techniques studied in this thesis, system call traces were collected from benign and malicious software samples over a six-month period. The traces were collected in home, research laboratory, and public computer laboratory settings, and on a custom testbed, where recently discovered malware samples were executed. To collect the traces, a custom host agent known as the System Call Service (SCS) was created. The main benefits of the SCS are that it is low-overhead, transparent to the user, and can be deployed on any host running a recent version of Microsoft Windows. This chapter describes the design and function of the SCS. It also describes the datasets collected using the SCS, which are used to evaluate the feature extraction, malware detection, and malware classification techniques described in Chapters 4-6.

3.1 System Call Service (SCS)

The SCS was created for two reasons. First, the specialized analysis tools used in the related work to collect system calls offer a limited view of software functionality and exhibit high enough overhead to preclude their usefulness in production environments. Second, alternative techniques used in the related work rely primarily on application programming interface (API) hooking or system service descriptor table (SSDT) hooking. The former technique requires the modification of running software to intercept function calls, while the latter requires the overwriting of a kernel data structure to intercept system calls. Both techniques affect the stability and reliability of the systems on which they are deployed, and their usefulness is limited in recent versions of Windows. In contrast to traditional system call analysis tools, the SCS

- uses built-in kernel logging functions available in recent versions of Windows;
- monitors access to every system call that is part of the core operating system (OS) kernel;
- does not require modification of monitored processes or kernel data structures; and

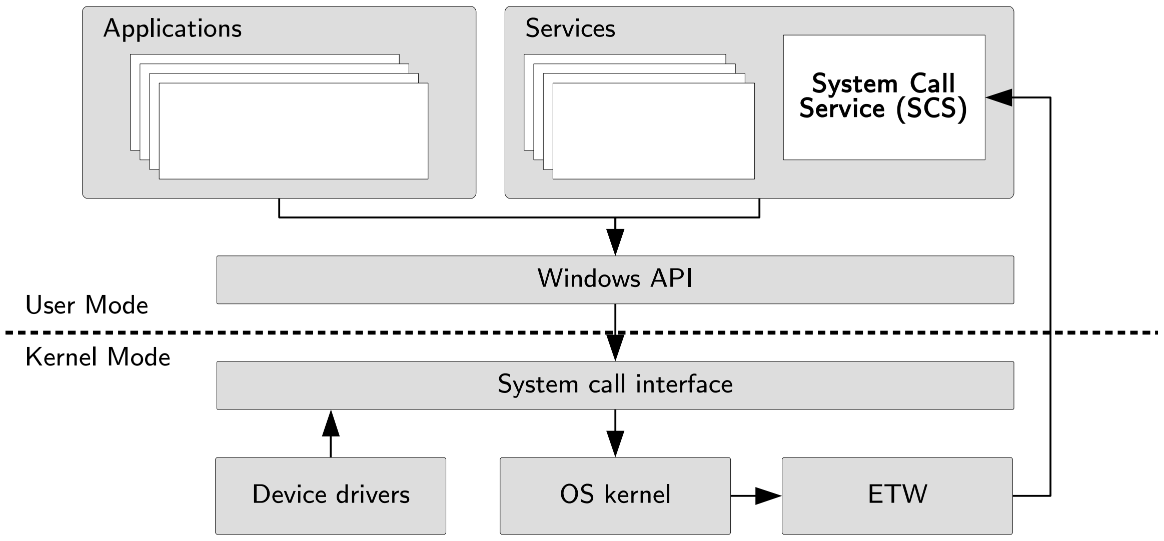

Figure 3.1: System Call Service (SCS) block diagram, showing how system calls originating from all processes, including applications, services and drivers, are captured by the SCS using the Event Tracing for Windows (ETW) framework

- exhibits low enough computational and memory overhead for deployment in production environments.

The block diagram presented in Figure 3.1 illustrates the function of the SCS. The arrows indicate the flow of system calls from applications, services, and drivers to the SCS where they are logged. System calls are made in user-mode by applications and services through the Windows API and in kernel-mode by device drivers and other code running at the kernel-level. The system calls are handled by the system call interface, and the functions requested by the system calls are executed by the OS kernel. The Windows kernel contains built-in kernel tracing functions provided by the Event Tracing for Windows (ETW) framework. The SCS runs as a service application on the monitored host.

The ETW framework is an event-driven framework. The events reported by the kernel tracer include process creation and deletion, thread creation and deletion, system call enters and exits, and context switches. The SCS tracks all of these events to attribute each system call to the process and thread from which it originated. An example of the kernel tracer events and their metadata is provided in Figure 3.2, which shows all of the system call enter, context switch, and thread start

```
System Call Enter          FFFFF801 54FB8EE0 on core 0
Context Switch             Exchanged 9224 4 for 0 0 on processor core 7
System Call Enter          FFFFF801 54CA7A30 on core 8
Context Switch             Exchanged 7788 5488 for 0 0 on processor core 8
System Call Enter          FFFFF801 54FB8EE0 on core 0
System Call Enter          FFFFF801 54FB8EE0 on core 0
Thread start               TID 8000 PID 5488
System Call Enter          FFFFF801 5507FD84 on core 0
Context Switch             Exchanged 0 0 for 8000 5488 on processor core 5
System Call Enter          FFFFF801 54FF92F0 on core 0
System Call Enter          FFFFF801 54F9977C on core 0
System Call Enter          FFFFF801 54FA5CB4 on core 0
System Call Enter          FFFFF801 54F9977C on core 0
System Call Enter          FFFFF801 54F9977C on core 0
System Call Enter          FFFFF801 54FB0D24 on core 5
System Call Enter          FFFFF801 54FF92F0 on core 0
System Call Enter          FFFFF801 5507FE8C on core 5
Context Switch             Exchanged 8000 5488 for 0 0 on processor core 5
System Call Enter          FFFFF801 54F9977C on core 0
System Call Enter          FFFFF801 54FA5CB4 on core 0
System Call Enter          FFFFF801 54FF92F0 on core 0
System Call Enter          FFFFF801 54FF92F0 on core 0
Context Switch             Exchanged 0 0 for 5380 5424 on processor core 19
```

Figure 3.2: Kernel trace using Event Tracing for Windows (ETW), showing system call enter, context switch, and thread start events over a $155\mu s$ interval

events collected over a $155\mu s$ interval. The system call events identify a system call by its memory address (a 64-bit hexadecimal number), and identify the logical processor core on which each system call originated. The SCS uses kernel debugging symbol tables to map the memory addresses to system call names. The processes and threads executing on each processor core are identified by the context switch events. The context switch events list the identifiers of the threads and processes switched out of and onto a processor core. The identities of the processes and threads are determined by monitoring the thread and process creation events.

Since the SCS tracks the threads and processes responsible for each system call, it can record system call traces at the host, application, process, and thread levels. For this study, only process level system calls are considered. The output of the SCS is a collection of system call traces, one for every process observed on the monitored host. To assist in post-mortem analysis, the SCS also records the following metadata for each host, process, and thread.

Host metadata

- Kernel version
- Processor cores
- Host identifier

Thread metadata

- Start time
- Stop time
- Thread identifier (TID)
- Parent process

Process metadata

- Start time
- Stop time
- Process identifier (PID)
- Parent process
- Image name
- Image path
- Image checksum (MD5)

The PID and TID identifiers are values used internally by the OS to identify threads and processes. The process image name, path, and checksum refer to the executable file from which the process was created, and are used to identify the software applications associated with each process. The host identifier is an anonymized identifier used to differentiate data collected from different hosts.

The SCS runs on both 32-bit and 64-bit versions of Windows, including Windows 7, 8, 8.1, Server 2008, Server 2008 R2, Server 2012, and Server 2012 R2. It is written in C# using the Microsoft .NET Framework version 4.5. The outputs of the SCS are a collection of files each containing the system call trace of a single process, and a database containing the described metadata. A representative dataset collected from running the SCS for 60 minutes on a production host included metadata from 90 processes and 3,255 threads. For this study, the SCS was configured to collect the first 10,000 system calls made by every process. The SCS stores the system call traces in a concise binary format, resulting in system call trace sizes of at most 20 KB.

3.1.1 Memory and computational overhead

To be useful for data collection in production environments, the SCS was designed to have minimal memory and processing overhead. The memory overhead of the SCS was determined by checking the size in memory of the SCS under a variety of different conditions, including different OS versions, .NET Framework versions, hardware configurations, and workloads. In total, the memory footprint of the SCS was tracked over the period of one hour on 12 different hosts. The average memory usage of the SCS process over the period of study was 50 MB. The SCS used more memory when the workload on a system was high, particularly when there were a large number of processes being simultaneously traced. In these circumstances, the SCS used as much as 170 MB of RAM. The maximum memory usage was observed on a 12-core host serving as a virtual machine manager (VMM) at nearly 100% CPU usage. Conversely, when the workload on a system was low and few processes were being traced, the SCS used as little as 10 MB of RAM.

The computational overhead of the SCS was evaluated using the benchmark program PCMark8[1]. PCMark8 was used to characterize the computational overhead by comparing the benchmark performance on a host running the SCS to the performance of the same host when the SCS was disabled. The following benchmarks were used for this study:

- web browsing, which loaded and navigated websites;
- writing, which composed and manipulated text documents;
- photo editing, which displayed and manipulated image files;
- gaming, which rendered 3D graphics using DirectX;
- video playback, which played high definition video; and
- video encoding, which transcoded high definition video.

The PCMark8 benchmark program was chosen because its benchmarks represent real-world use cases, and because it provides timing information for individual benchmarks. The baseline performance was established by running the benchmarks on a host with the SCS disabled. For the web browsing, writing, video encoding, and photo editing benchmarks, the overhead was measured as the percentage increase in time it took to complete the benchmark over the baseline. For the

[1]PCMark8, FutureMark, `http://www.futuremark.com`

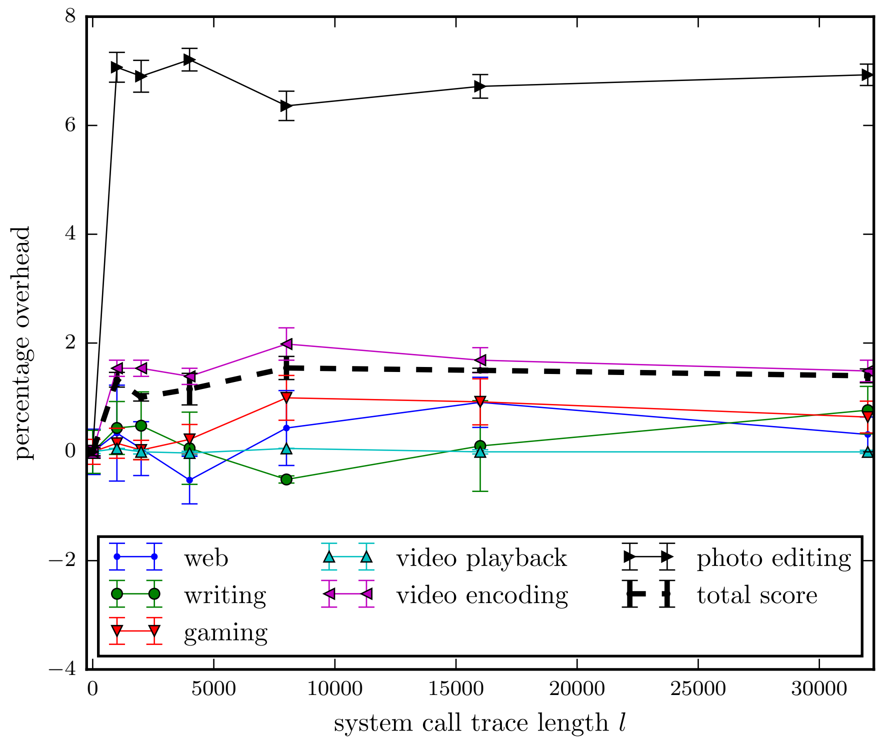

Figure 3.3: Overhead analysis of the System Call Service (SCS) on a host in a home environment

gaming and video playback benchmarks, the overhead was measured as the percentage decrease in the frame rate from the baseline. PCMark8 also provides an overall score, a weighted sum of the individual benchmarks, that characterizes the overall performance of a system. The overhead for the score was computed as the percentage decrease in the score.

Figure 3.3 summarizes the results of the benchmarks for a host in a home environment. The figure shows the overhead of each of the benchmarks versus l, the maximum number of system calls the SCS recorded from each process. The results presented are the average results from nine trials, and the error bars show standard error of the mean (SEM) of the results. The results presented for $l = 0$ are the baseline results achieved when the SCS was disabled.

Figure 3.3 shows that the SCS had no measurable effect on the video playback benchmark, indicated by the near-zero overhead for all trace lengths. Video playback runs as a single process and is neither processor-intensive nor system-call intensive. For the majority of the other benchmarks, the overhead increased with the trace length. For each benchmark, the overhead reached its maximum value once the trace length was long enough to record the majority of the system calls made during the execution of the benchmark. The web, writing, gaming, and video encoding benchmarks all experienced a maximum of 2% overhead. The photo editing benchmark experienced 7% overhead, independent of the length of the collected traces. The dashed line in Figure 3.3 indicates the overhead as measured by the overall benchmark score, indicating a maximum overhead under 2%.

The processing and memory overhead introduced by the SCS is largely dependent on the type and intensity of the system load. In particular, short-lived processes that are CPU and system call intensive experience the highest overhead, incurred primarily during process creation and deletion. The overhead of the SCS results primarily from its input/output (I/O) routines, since the SCS logs process creation and deletion events and writes system call traces directly to disk. Furthermore, the system call trace for each process is stored as a separate file, introducing file creation overhead at the start of each trace and writing overhead at the end of each trace. These factors were largely responsible for the high overhead observed for the photo editing task, which was both processor intensive and involved the execution of many short-lived processes.

The SCS performs so much I/O because it is primarily a data collection tool, designed to record raw system call traces. Raw system call traces were collected to study the effects of feature extraction strategies, but would not be collected in a production deployment of the system. Rather, the SCS would store aggregate information about the system call traces in memory and perform detection and classification on the aggregate data. Thus, the I/O burden of the SCS would decrease. Furthermore, the overhead of the SCS could likely be decreased by optimization of its source code or by implementing critical features in a lower-level language.

3.2 Production data collection

The goal of this work is to evaluate the accuracy of a malware detection and classification system against real-world datasets. To achieve this goal, the SCS was deployed in production environments in a research laboratory, in a home environment, and in public computer laboratories. The main purpose of this deployment was to collect traces from a variety of benign software programs against

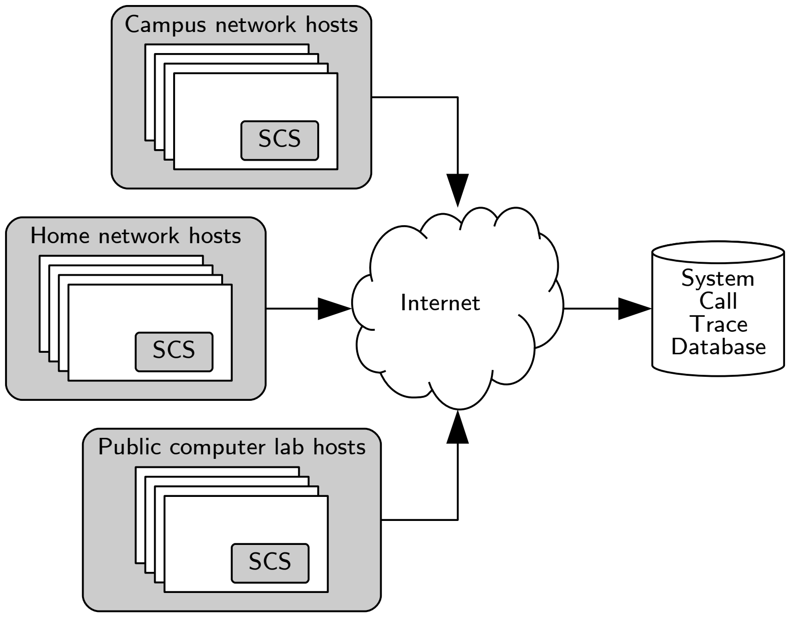

Figure 3.4: System Call Service production deployment, showing the home, campus, and public computer lab hosts sending system call traces to a centralized system call database

which to evaluate the detector. Figure 3.4 shows a diagram of the production deployment, indicating the three different types of hosts where the SCS was deployed. The SCS recorded system call traces on the monitored hosts and periodically transmitted the data over the Internet to a centralized server, where the system call traces were stored in a database for later processing. The centralized storage and processing of system call traces was performed for research purposes only. For a production deployment, the SCS would perform local collection of the traces, which would be processed as they are collected.

The research laboratory and home environments included 14 hosts with a variety of different configurations and uses. The hardware included laptops, desktops, and servers, with $1 - 24$ logical processor cores and $2 - 32$ GB of RAM. The host OSs included Windows 7, 8, Server 2008 R2, and Server 2012 R2. The hosts were used as workstations for desktop publishing, technical drawing, scientific computing, web-browsing, gaming, and multimedia. Two of the hosts were servers used as the VMM for a malware detection testbed. The data from these environments were used for model development and to characterize the false positive rate (FPR) of the detector.

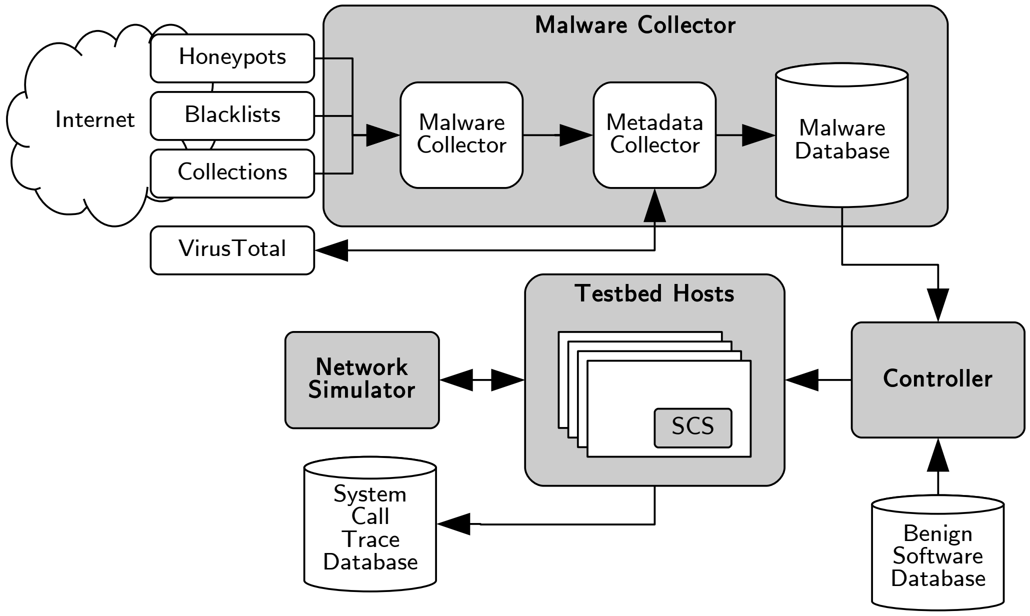

Figure 3.5: Malware testbed block diagram

The SCS was deployed in two public access computing sites on 10 hosts. These computing sites were part of KEYSPOT Network, led by The City of Philadelphia's Office of Innovation and Technology (OIT), The Mayor's Commission on Literacy, and The Department of Parks and Recreation (PPR). The hosts were used by the public primarily for web-browsing and desktop publishing. They were also used by students attending courses in technical literacy and professional development. Users were provided a standard suite of pre-installed software and given permission to install additional software at their discretion. The hosts were workstations running 32-bit and 64-bit editions of Windows 7. The data from the computing sites were used in a case study to determine how well the developed techniques and models generalized to new environments.

3.3 Malware testbed

The malware testbed is a collection of 19 computer hosts created specifically for studying malware behavior. The malware testbed was created to address the shortcomings of traditional malware analysis environments, namely that they limit execution paths and monitor only a small fraction of available system calls. More importantly, the malware testbed was created for consistency, so malware could be studied using the same set of tools and in the same environments as the benign software. This was done to ensure that the tools and environments did not bias the study, and to ensure that the developed techniques could be implemented in production environments with comparable results. Figure 3.5 provides a block diagram of the malware testbed, indicating its four primary components (highlighted in gray):

- a malware collector, which gathers malware samples to use for analysis;
- a controller, which coordinates the analyses of the malware samples;
- the testbed hosts, where the malware samples are executed; and
- a network simulator, wcich interacts with the malware samples.

3.3.1 Malware collector

The malware collector was used to gather malware samples to study on the testbed, with a focus on collecting currently active malware threats. It collected malware by deploying honeypots, crawling blacklisted websites, and importing existing threat collections. Honeypots are decoys used to attract cyber-attacks and were used to collect the payloads delivered by such attacks. The malware collector used the Dionaea[2] honeypot as part of the Stratagem[3] Linux distribution to collect such payloads. URL blacklists are commonly used by web browsers and security software to prevent users from visiting malicious URLs. The threat collector intentionally visited these URLs, scraping the websites for suspected malware samples. Finally, the threat collector imported malware collected by other means, including malware samples obtained from publicly available malware collections and manually collected malware samples.

The metadata collector was used to collect information about each malware sample, including its hash, file type, origin, target platform, and matching antivirus (AV) definitions. The metadata collector used the free online virus scanning service VirusTotal[4] to scan each incoming malware sample against multiple commercial antivirus (AV) definitions, and used the file identification tools

[2]Dionaea low-interaction honeypot, `http://dionaea.carnivore.it`
[3]Stratagem honeypot distribution, `http://sourceforge.net/projects/stratagem/`
[4]VirusTotal, `http://www.virustotal.com`

`TrID` and `file` to identify file types and target platforms. The collected metadata were stored in a database along with the malware samples. For each malware sample, the collected metadata included

- the number of AV detectors with which the sample was scanned,
- the number of positive detections,
- the AV labels of the sample,
- the date the sample was added to the collection,
- the source from which the sample was obtained,
- the date the AV results were last updated,
- the date the sample was first seen by VirusTotal,
- `file` and `TRiD` results, and
- `MD5` and `SHA1` hashes of the sample.

3.3.2 Controller and virtual machine hosts

The controller had two primary tasks, setting up the virtual machine (VM) hosts and injecting malware. The controller interfaced with the Microsoft Hyper-V VMM to manage the VM images, and used Microsoft PowerShell remoting to interface with the VM hosts. For each malware sample, the controller performed the following sequence of tasks.

1. Restore a VM to a known clean state.
2. Transfer a malware sample onto the VM.
3. Start the VM with the SCS enabled.
4. Execute benign software on the VM at randomly determined time instances.
5. Execute the malware sample on the VM.

6. Shut down the target VM.

7. Transfer the collected system call traces and metadata to a database for analysis.

The VM hosts

A challenge in executing malware samples is that, in general, the dependencies of the malware samples are not known *a priori*. For example, a malware sample may require specific OS versions, application frameworks, or target applications to perform its malicious tasks. To overcome this challenge, multiple VM host configurations were employed on the testbed, including hosts configured specifically for the testbed and hosts cloned from production environments. The hosts ran multiple Windows OS versions, had different sets of software installed, and had varying levels of software patches applied. This diversity was used to make it more likely that a malware sample would execute successfully on at least one of the hosts in the test environment.

The malware testbed consisted of two physical servers, each running Windows Server 2012 R2 and Microsoft Hyper-V. The 19 VM hosts used for this study ran Windows 7 (32-bit and 64-bit editions), Windows 8, Windows 8.1, and Windows Server 2012 R2. The VM configurations each included $1 - 12$ processor cores and $2 - 8$ GB of RAM. The software installed on each host included multimedia editing software, programming utilities, a multimedia server, and desktop publishing software.

Benign software

To mitigate potential bias that could arise from running only malware on the testbed hosts, the controller executed both benign and malicious software on the hosts. More than 2,000 benign applications were considered, including

- software distributed with the OS;
- video games, including first-person-shooters and card games;
- audio, video, and graphics viewing and editing software;
- office and productivity tools;

- security-related tools, including AV software;
- Internet related software, such as web browsers, videoconferencing, torrent, and messaging software; and
- system utilities, such as file management and registry editing tools.

The benign software were instrumented in two different ways; either the software were allowed to run without any interaction, or application-specific, scripted tasks were performed. The former approach was used to mimic the method by which malware was injected, while the latter was used to mimic more realistic use. Scripted tasks included saved video games being replayed, system monitoring and management tools performing predetermined actions, software being installed, and benchmarks being executed. The benign software programs and OSs installed on the testbed were also periodically updated to their latest versions during the data collection period.

Malware injection

The second primary task of the controller was the injection of malware samples onto the hosts. This task was performed in the same manner that the benign software were executed, using PowerShell remoting. Each VM instance was used to execute only one malware sample. Each sample was executed at a randomly determined time after the VM was started, between zero and fifteen minutes, and allowed to run for at least four minutes before the VM was terminated. Each malware sample was executed only once on the testbed.

3.3.3 Network simulator

A challenge in characterizing malware behavior is allowing the malware to perform their malicious functions as they would in a production environment without putting production systems at risk. To minimize the risk to production systems, the VMs used in this study were connected to an isolated network. However, many malware samples perform tasks that require Internet connectivity. Such tasks include communicating with command and control servers, propagating to other vulnerable hosts on the network, transmitting stolen information, and sending spam.

To address this concern, the VMs were connected to an isolated network configured to route all traffic from the VMs to a network simulator. The network simulator ran the Strategem Linux distribution and a DNS server, configured to resolve all domain queries to the address of a Dionaea honeypot. The honeypot configuration mirrored the configuration of the honeypot used for malware collection (see Section 3.3.1). The honeypot provided network interaction with the malware, enabling the malware samples to perform some of their network-centric tasks without exposing the malware to a real-world network.

Over the course the described malware testing effort, the honeypot collected

- tens of thousands of logged TCP sessions per day;
- malicious binaries dropped by the malware samples executing on the testbed;
- detailed information collected by the malware samples about the testbed hosts, including hostnames, IP addresses, license keys, passwords, and keylogger data;
- notifications from the malware samples indicating when a testbed host had been infected; and
- compressed or encrypted data files whose content could not be determined.

3.4 Malware samples

The malware collector described in Section 3.3.1 was used to collect more than 750,000 distinct suspected malware samples, all standalone portable executable (PE) files. For this study, only those samples first discovered after 1 January 2012 and identified as malicious by at least fifteen different AV vendors were chosen. The first criterion was used to ensure that only recent malware samples were used in the study, and the second criterion was used to ensure that the suspected malware samples were actually malicious.

Of the malware samples studied, only a fraction successfully executed on the testbed. The execution of a malware sample was considered successful if the processes created by the sample ran for the duration of the data collection and performed at least 1,500 total system calls. These criteria were established to ensure that enough information was collected from each malware sample

Table 3.1: Most common malware categories used in this study, according to their Microsoft and Kaspersky labels

Microsoft category	samples
Backdoor	8,989
Worm	7,095
Virus	6,501
Trojan	6,371
Password stealer (PWS)	3,172
VirTool	3,105
TrojanDownloader	2,445
TrojanDropper	1,420
Rogue	996
TrojanSpy	713

Kaspersky category	samples
Trojan	16,867
Backdoor	7,678
Virus	6,888
Worm	2,339
Trojan-Dropper	2,151
Trojan-Spy	2,048
Email-Worm	1,744
Trojan-Downloader	1,272
Trojan-Ransom	1,158
Net-Worm	1,029

to perform the analyses presented in this thesis. More than 55,000 distinct malware samples met these criteria and were used in this study. Some 9,000 samples were executed on the testbed, but did not produce enough data to be considered for the study. The majority of these samples terminated immediately upon execution, likely because the host did not meet their requirements. Some 2,000 samples were injected on the testbed but failed to execute.

Among the malware samples used in this study, Table 3.1 shows the 10 most common malware categories, according to their Microsoft and Kaspersky labels. The table shows the number of malware samples used in this study that belong to each of the categories. In total, the malware samples represented 26 Microsoft categories and 25 Kaspersky categories. The most common categories were Trojans, backdoors, viruses, worms.

Section 1.1 described two shortcomings of malware labeling systems, namely their incompleteness and inconsistency. Table 3.1 and the data from which it was created provide additional evidence of these shortcomings. First, nearly 25% of malware samples used in this study were unlabled by the Microsoft naming scheme, and nearly 15% were unlabeled by Kaspersky, indicative of the incompleteness of each of these naming systems. Furthermore, a comparison number of samples in each category between the two tables illustrates the inconsistency of the naming schemes. For example, nearly 50% of the labeled malware samples were identified as some type of Trojan by Kaspersky, whereas only 25% of the labeled samples were identified as such by Microsoft.

Table 3.2 shows the 20 most common malware families among the malware samples used in this

Table 3.2: Most common malware families used in this study, according to their Microsoft and ESETNOD32 labels

Microsoft family	instances
Worm.Mydoom	1,350
Backdoor.Fynloski	1,326
VirTool.CeeInject	1,201
Trojan.Malex	1,155
Backdoor.Bladabindi	1,132
Backdoor.Hupigon	1,082
Backdoor.Kelihos	1,041
Worm.Gamarue	1,010
Rogue.Winwebsec	877
PWS.OnLineGames	856
Worm.Allaple	762
Virus.Almanahe	756
Virus.Parite	711
TrojanDropper.Loring	703
Virus.Madang	699
VirTool.Obfuscator	647
Virus.Ramnit	644
Worm.Vobfus	632
Backdoor.Simda	617
Backdoor.Zegost	616

ESET family	instances
Kryptik	4,799
Injector	4,765
Sality	1,990
Delf	1,721
AdWare.MultiPlug	1,530
MSIL/Injector	1,494
Mydoom	1,323
Agent	1,298
Fynloski	1,068
Hupigon	1,000
PSW.OnLineGames	982
Ramnit	961
IRCBot	949
MSIL/Bladabindi	816
Spy.Zbot	762
Alman	751
Madang	702
Farfli	662
Parite	652
VB	630

study, using the family names derived from their Microsoft and ESET labels. The tables show the number of malware samples used in this study that belong to each of the families. In total, there were 1,373 Microsoft families and 1,012 ESET families represented in the malware set. The table indicates that many common and well-defined malware families, such as MyDoom, Zeus/ZBot, and Ramnit were among the most represented families used in this study. Other families listed in Table 3.2, such as VirTool.Obfuscator and Injector, are more generically defined. VirTool.Obfuscator refers to any generic obfuscated malware, and Injector refers to any malware that has the capability to inject code into another process.

3.5 Ground truth labeling

The term "ground truth" refers to the absolute truth of something, and is used here to refer to the assumed correct labels applied to the system call traces collected for this study. Although the ground truth is treated as absolute, knowing whether every process is benign or malicious with certainty is impossible. However, it is expected that the occurrence of errors in the ground truth are rare, thereby limiting the effects that incorrect ground truth have on the experimental results. Applying ground truth labels to malware samples for classification presents another challenge altogether, as the existing AV labeling systems are incomplete an inconsistent. As such, this study considered multiple ground truth labeling schemes derived from AV labels.

The binary ground truth labels indicated whether a process was benign or malicious. For malware classification, all of the processes labeled 'malware' were also given ground truth labels based on their AV labels. The malware traces were labeled according to 27 different labeling schemes derived from AV labels, including category and family labels. For each malware sample executed on the testbed, all of the processes created by the sample were assigned the same label.

The binary ground truth was generated primarily by comparing the checksums of the executable images from which the processes were created against the VirusTotal database. While this covered the majority of the processes, there were others that were not in the VirusTotal database. These processes either had their images submitted to VirusTotal for analysis or were labeled based on the circumstances surrounding their execution. For example, unknown processes created after the execution of a malware sample on the testbed were labeled 'malware'. Among these were apparent metamorphic or polymorphic malware samples, which spawned hundreds of processes, each with a distinct checksum. Unknown processes created before the execution of malware sample or observed in the laboratory or home environments were labeled 'benign'.

3.6 Conclusions

The SCS was used for a six month data collection campaign that resulted in the collection of some 1,000 host-days of data from 43 hosts. Of the data that were collected, this thesis studies the set of all processes whose system call traces contained at least 1,500 system calls. This restriction enables accurate comparisons of detection performance on trace lengths up to 1,500 calls. The datasets used for the experimental evaluation presented in this thesis include more than

- 135,000 system call traces from the execution of more than 55,000 distinct malware samples on the tested;

- 4 million benign process traces from the execution of more than 7,000 distinct benign executable images; and
- 56,000 system call traces collected from public computer labs.

The collected data are used along with their ground truth labels to evaluate the effectiveness of the feature extraction, malware detection, and malware classification techniques described in the following chapters.

Chapter 4: Feature Extraction

This chapter addresses the first of the three major challenges presented in this thesis, that of feature extraction. Feature extraction is the process of representing raw data as a set of features that are informative and that facilitate analysis. This chapter explores feature extraction techniques primarily inspired by document classification and adapted to work with the system call traces collected by the System Call Service (SCS) described in Section 3.1. This chapter divides the feature extraction process into four components,

Information retrieval: the process of extracting information from raw system call traces to use for detection and classification (Section 4.1);

Feature selection: the process of selecting the most informative subset of features (Section 4.2);

Feature scaling: the process of applying weights to the extracted features to facilitate analysis (Section 4.3); and

Feature reduction: the process of reducing the number of dimensions of the feature data to remove redundancy and facilitate analysis (Section 4.4).

Figure 4.1 depicts the end-to-end feature extraction process as it is performed in practice, indicating the primary transformations considered for each of the four components. The feature extractor takes as its input a system call trace from the SCS and outputs the representation of the trace as a numeric feature vector $\hat{\mathbf{x}}$. The output feature vector $\hat{\mathbf{x}}$ is used as the input to the malware detection and classification components of the system. The objective of this study is to identify the subset of feature extraction techniques that afford the best detection and classification accuracy. The remainder of this chapter provides detailed descriptions of the techniques considered for each of the four components.

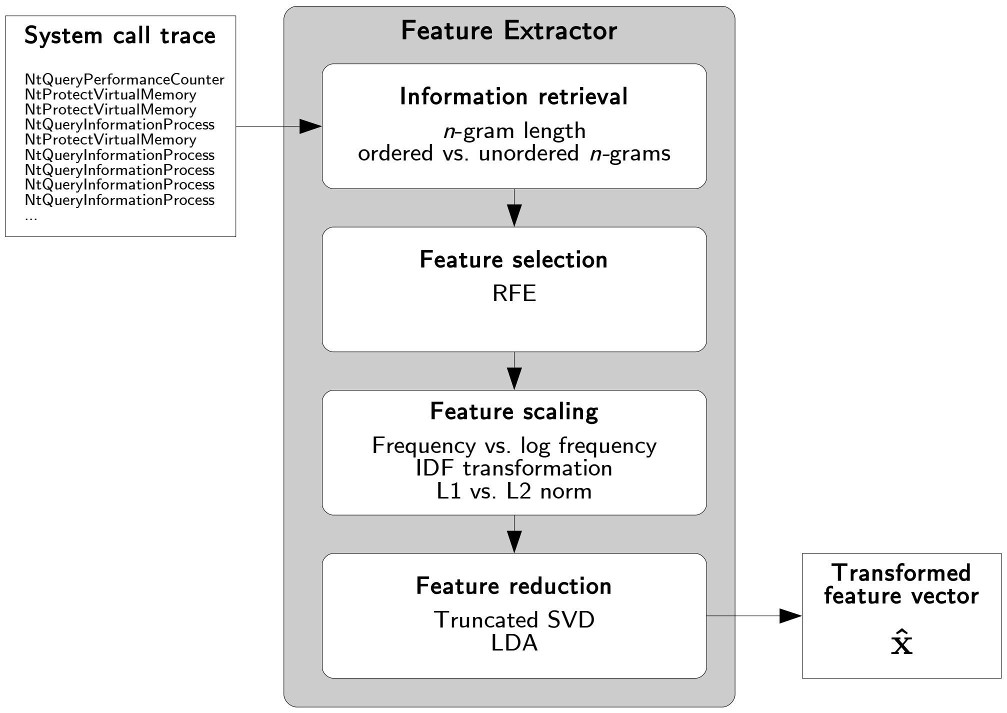

Figure 4.1: Feature extraction process, showing each of the four stages and the the transformations performed at each stage

4.1 Information retrieval

Previous work in system call-based malware detection and classification identified informative features, including call frequencies, ordered call sequences, and unordered call tuples (Section 2.1.3). This thesis presents a comparative study of these three types of features, using feature extraction techniques inspired by research in document classification. Document classification is the process of identifying to which category a text document belongs. The similarity between the system call traces used in this thesis and the datasets used for document classification is the structure of the data. In document classification, a text document can be represented as an ordered sequence of words. Similarly, a system call trace can be represented as an ordered sequence of system calls. In document classification, the patterns of words can be used to identify one of many characteristics, including the topic, tone, or author of the text. In malware analysis, the patterns of system calls can be used to determine the function of a program [39].

Feature extraction begins with information retrieval, which is the process of extracting informa-

Table 4.1: Ordered and unordered 2-gram representation of a system call trace

system call 1	system call 2	ordered	unordered
NtAllocateVirtualMemory	NtFreeVirtualMemory	1	2
NtAllocateVirtualMemory	NtQuerySystemInformation	1	2
NtAllocateVirtualMemory	NtQueryVirtualMemory	1	1
NtClose	NtAllocateVirtualMemory	1	1
NtCreateSection	NtMapViewOfSection	1	1
NtFreeVirtualMemory	NtAllocateVirtualMemory	1	–
NtFreeVirtualMemory	NtOpenDirectoryObject	1	1
NtMapViewOfSection	NtQuerySection	1	1
NtOpenDirectoryObject	NtOpenDirectoryObject	1	1
NtOpenDirectoryObject	NtOpenSymbolicLinkObject	1	1
NtOpenFile	NtCreateSection	1	1
NtOpenFile	NtQueryVolumeInformationFile	1	1
NtOpenSymbolicLinkObject	NtQuerySymbolicLinkObject	1	1
NtProtectVirtualMemory	NtProtectVirtualMemory	1	1
NtProtectVirtualMemory	NtQueryInformationProcess	2	3
NtQueryAttributesFile	NtOpenFile	1	1
NtQueryInformationProcess	NtProtectVirtualMemory	1	–
NtQueryInformationProcess	NtQueryInformationProcess	3	3
NtQueryInformationProcess	NtQuerySystemInformation	1	1
NtQueryPerformanceCounter	NtProtectVirtualMemory	1	1
NtQuerySymbolicLinkObject	NtClose	1	1
NtQuerySystemInformation	NtAllocateVirtualMemory	1	–
NtQuerySystemInformation	NtFreeVirtualMemory	1	1
NtQuerySystemInformation	NtQuerySystemInformation	1	1
NtQueryVirtualMemory	NtOpenFile	1	1
NtQueryVolumeInformationFile	NtQueryAttributesFile	1	1
total		29	29

tion from the raw system call traces. For information retrieval, a bag-of-words model is used to represent the traces. In text classification, a bag-of-words model is a representation of a text document as a vector of word frequencies; *i.e.*, a vector encoding the number of occurrences of each word appearing in the document [107]. For malware analysis, a bag-of-words model is a representation of system call trace as a vector of system call frequencies.

Representing a system call trace as a vector of system call frequencies is useful because the numeric feature vectors facilitate malware detection and classification. However, system call frequencies alone are not particularly informative (Section 5.4.1). Instead, this study also uses n-gram analysis for information retrieval, where an n-gram is a contiguous ordered sequence of n items extracted from from a larger sequence. The technique of n-gram analysis has applications in document classification and computational linguistics, where n-grams are sequences of letters or words extracted from a text [108, 109]. System call n-gram analysis has proven useful for intrusion detection when paired with a signature-based detector (Section 2.1.3).

This thesis considers two different types of system call n-grams. *Ordered* n-grams are ordered sequences of n system calls appearing contiguously in a trace. Ordered n-grams provide information about the local ordering of system calls. *Unordered* n-grams are unordered sets of n system calls appearing contiguously in a trace. Unordered n-grams provide information about which system calls typically appear near each other, but do not consider the order in which the calls appear. Both types of n-grams are extracted from system call traces using a sliding window approach. For this study, the bag-of-words representation and n-gram analysis are combined, and each system call trace is represented as a vector of n-gram frequencies. This information retrieval technique is referred to as the bag-of-n-grams model.

An example bag-of-n-grams representation of a system call trace is shown in Table 4.1. The table shows both the ordered and unordered 2-gram representations of the system call trace in Figure 1.1 in Section 1.3. The table lists every distinct sequence of two contiguous system calls appearing in the trace, along with the frequency of occurrence of each. Whereas the ordered 2-gram representation differentiates between the sequences $\langle$ NtFreeVirtualMemory, NtAllocateVirtualMemory $\rangle$ and $\langle$ NtAllocateVirtualMemory, NtFreeVirtualMemory $\rangle$ and considers them to be distinct 2-grams, the unordered 2-gram representation considers these sequences to be instances of the same 2-gram, $\{$ NtAllocateVirtualMemory, NtFreeVirtualMemory $\}$. The '–' symbol indicates that a sequence is a reordering of another sequence listed in the table.

While Table 4.1 shows only the system call sequences appearing in the trace in question, a system call trace can be more generally represented as a column vector $\mathbf{x}$ with one entry for each of the possible system call n-grams. The SCS, which was used to collect the system call traces used in this study, monitors access to 465 distinct system calls. Therefore, the length of the vector $\mathbf{x}$ is 465^n for the ordered n-gram representation, and $O(465^n)$ for the unordered n-gram representation. For a system call trace of length l, where $l \ll 465^n$, $\mathbf{x}$ is sparse, *i.e*, the vast majority of the features in the vector are zero. To take advantage of this sparsity, the feature vectors $\mathbf{x}$ are stored in a sparse array format when $n > 1$. Therefore, the storage complexity of a system call trace is $O(l)$. The techniques described in the following chapters, especially the stochastic gradient descent (SGD) algorithm used to train the support vector machine (SVM) and logistic regression (LR) detectors and classifiers, were selected to take advantage of the sparsity of the feature data.

4.2 Feature selection

The system call n-gram feature space is a very high dimensional space when $n > 1$. Furthermore, it is not likely that every n-gram is equally informative for detection and classification. For example, the frequency of occurrence of an article (such as 'the') in a text document likely conveys little information about the topic of the document. Similarly, the frequency of occurrence of a commonly used system call (such as NtAllocateVirtualMemory) likely conveys little information about the malicious function of a process. The goal of feature selection is to experimentally identify the subset of n-grams that is most informative.

The motivation for feature selection is two-fold. First, the storage and processing of high-dimensional datasets introduces memory and processing overhead. The memory overhead arises from the storage of the feature vectors, and the processing overhead arises from the application of feature extraction, detection, and classification algorithms. Minimizing such overhead is especially important in this study, because detection and classification are to be performed online and in real-time on production hosts. The number of features grows exponentially in n, causing a dramatic increase in overhead as n increases. Second, including non-informative features during training can lead to overfitting, degrading detection and classification accuracy. Overfitting occurs during training when a detector or classifier erroneously includes uninformative features in its decision rule.

In this work, recursive feature elimination (RFE) was used for feature selection [110]. RFE was chosen over alternate feature selection algorithms for three primary reasons: First, RFE is a multivariate technique that considers the set of all features during feature selection. This contrasts with univariate techniques, which consider each feature in isolation. The advantage in using a multivariate technique is that relationships among distinct features are considered during feature selection. Next, RFE performs feature selection by evaluating a detector or classifier's accuracy against training data. Thus, the subset of features selected by RFE is the subset that maximize the empirical accuracy of the detector or classifier used for analysis. Finally, RFE works with detection algorithms that provide feature weights. This is especially useful because the detection algorithms considered in this work that provided the highest detection accuracy provide feature weights. Those two algorithms, SVM and LR, were used to perform RFE.

RFE consists of both a training algorithm and a testing algorithm. The training algorithm determines the appropriate subset of features to use for analysis, and the testing algorithm eliminates the uninformative features from consideration. The testing algorithm selects the appropriate subset of features from the vector $\mathbf{x}$. In practice, information retrieval and feature selection are combined during testing so that the SCS collects only information about the selected n-grams.

The RFE training algorithm works by training the detector or classifier with the set of all features and removing the feature(s) with the lowest weights from consideration. This process is repeated recursively until the desired number of features is selected. Since detection and classification are part of the RFE training process, the RFE training algorithm also performs feature scaling. Described in detail in the following section, feature scaling is a necessary step for preparing the data for detection and classification. Using the same feature scaling techniques for testing and training also ensures that the features selected during training maximize the testing performance. The inputs to the training algorithm are the set $\mathcal{X}$ of training instances, a vector $\mathbf{y}$ of the corresponding labels of the training instances, and the desired number of features. The output of the training algorithm is the selected subset of the original features. The RFE training process is summarized as Algorithm 1.

Algorithm 1 is useful if the desired number of features is known *a priori.* To determine the number of features to use for detection, RFE can be combined with cross-validation. Cross-validation is a model validation technique used here to assess the effectiveness of the detection or classification techniques used with RFE. Cross-validation is the process of partitioning a dataset into disjoint subsets, using one subset for training and the other for testing. During RFE, cross-validation is performed in two distinct steps. First, the training set is used train the detector or classifier. Then, the testing set is used to evaluate the performance of the detector or classifier. Finally, the features

Algorithm 1 Recursive feature elimination (RFE) training algorithm

```
desired_number ← the desired number of features
$\mathcal{X}$ ← the training data
$y$ ← the labels of the training data
selected_features ← RFE($\mathcal{X}$)

function RFE($\mathcal{X}$)
    if the number of features in $\mathcal{X}$ == desired_number then
        return selected feature subset
    else
        Perform feature scaling on $\mathcal{X}$ to get $\mathcal{X}_{\text{fs}}$
        Train the detection algorithm using the data $\mathcal{X}_{\text{fs}}$ and labels $\mathbf{y}$ to get the weights $\mathbf{w}$
        $\mathcal{X}_{\text{reduced}}$ ← the input $\mathcal{X}$ with the features with the lowest weight $\mathbf{w}$ removed
        return RFE($\mathcal{X}_{\text{reduced}}$)
    end if
end function
```

with the lowest weights are removed, and the process is repeated. The modified RFE training algorithm with cross-validation is summarized as Algorithm 2.

Algorithm 2 Recursive feature elimination (RFE) training algorithm with cross-validation

```
$\mathcal{X}$ ← the training data
$y$ ← the labels of the training data
scores ← an empty array to store the cross validation scores
RFECV($\mathcal{X}$)

function RFECV($\mathcal{X}$)
    Partition $\mathcal{X}$ and $y$ into training and testing sets
    Perform feature scaling and train the detector using the training set to get $w$
    Perform feature scaling and detection on the testing set
    Evaluate the accuracy of the detector against the testing set
    Prepend the accuracy to be beginning of the scores array
    $\mathcal{X}_{\text{reduced}}$ ← the input $\mathcal{X}$ with the feature with the smallest weight $\mathbf{w}$ removed
    RFECV($\mathcal{X}_{\text{reduced}}$)
end function
```

The RFE with cross validation training process is performed over multiple folds, and the accuracy scores are averaged over the folds. The selected number of features is that which provides the maximum cross-validation score,

$$\text{desired number of features} = \arg\max(\text{scores}). \tag{4.1}$$

After the desired number of features is selected, the RFE training process in Algorithm 1 is repeated with the desired number as input to identify the appropriate feature subset to use for detection or classification.

4.3 Feature scaling

Feature scaling is used to facilitate the application of detection and classification algorithms that are sensitive to the ranges of the feature data. The motivation for feature scaling in this work is that system call frequencies scale to different orders of magnitude. For example, in a subset of 6×10^9 system calls collected by the SCS, there were

- 4.3×10^8 calls to NtQueryVirtualMemory,
- 1.9×10^5 calls to NtRaiseHardError, and
- 19 calls to NtShutdownSystem.

The feature scaling transformations described in this section were selected to address the issue of varying feature scales.

4.3.1 TF-IDF transformation

Term frequency – inverse document frequency (TF-IDF) transformation is an information retrieval technique commonly used in document classification and also demonstrated to be useful in intrusion detection [77, 111]. In document classification, frequently occurring words (such as 'the') are not particularly informative, whereas rare words tend to carry a lot of information about the content of a document. To address this issue, TF-IDF transformation applies weights to the word frequencies, placing higher emphasis on rare words. Analogously, it is likely that frequently occurring system calls (such as NtQueryVirtualMemory) are similarly uninformative. Therefore, TF-IDF transformation is used to apply weights to system call sequences inversely to their frequency of appearance.

The TF-IDF transformation of a vector $\mathbf{x}$ is the element-wise product of two vectors: the term frequency (TF), computed directly from the feature vector $\mathbf{x}$, and the inverse document frequency

(IDF), computed from the set $\mathcal{X}$ of all feature vectors,

$$\text{TF-IDF}(\mathbf{x}, \mathcal{X}) = \text{TF}(\mathbf{x}) \odot \text{IDF}(\mathcal{X}) \tag{4.2}$$

TF transformation can be applied directly to the feature data without any training. However, IDF transformation requires training to learn the weights to apply to the features. This work considers two different approaches for computing the TF. In the first approach, the TF is the raw n-gram frequency,

$$\text{TF}(\mathbf{x}) = \mathbf{x}\,. \tag{4.3}$$

This approach does not perform any feature scaling in the TF, instead relying on subsequent steps to perform feature scaling. The second approach uses the logarithmic frequency of the n-grams as the TF. This is done to account for the varying orders of magnitude of the features. With the logarithmic frequency, Laplace smoothing is used to address the sparsity of the feature vectors [112], resulting in the following expression,

$$\text{TF}(\mathbf{x}) = \log(\mathbf{x} + 1)\,. \tag{4.4}$$

Like the logarithmic frequency, the IDF addresses the varying feature scales. It applies feature weights that vary inversely with frequency of occurrence of each feature, thereby placing higher emphasis on rare n-grams. Considering N to be the number of traces in the training set $\mathcal{X}$ and $\mathbf{d}$ to be a vector counting the number of traces in $\mathcal{X}$ in which each n-gram appears, the IDF is

$$\text{IDF}(\mathcal{X}) = \log\left(\frac{1+N}{1+\mathbf{d}}\right) + 1\,. \tag{4.5}$$

Laplace smoothing is used in the IDF calculation to prevent zero divisions arising from the sparsity of the data, and the result is offset by one to ensure that every feature receives a non-zero weight, even if it appears in every trace in the training set $\mathcal{X}$.

4.3.2 Unit-magnitude scaling

TF-IDF transformation addresses the issue of the varying orders of magnitude of the feature data using logarithmic frequency and IDF transformation. However, many machine learning algorithms (such as SVM and LR) perform best when operating on more rigidly scaled features. To address this issue, the TF-IDF transformed vectors are scaled to unit magnitude. For this work, both the L_1 (Manhattan) and L_2 (Euclidean) norms are considered. Scaling to unit magnitude was chosen over alternatives, such as rescaling or standardization, because it preserves the scale relationships among the system calls. Thus, the complete feature scaling process, where $\| \cdot \|$ indicates either the L_1 or L_2 norm, is

$$\mathbf{x}_s = \frac{\text{TF-IDF}(\mathbf{x}, \mathcal{X})}{\| \text{TF-IDF}(\mathbf{x}, \mathcal{X}) \|} \, . \tag{4.6}$$

4.4 Feature reduction

Feature reduction shares a common goal with feature selection; namely, to reduce the overhead of the system. While feature selection accomplishes this by removing uninformative features, feature reduction does so by identifying a non-redundant feature set. Feature reduction also has the benefit of reducing the effect of the so-called *curse of dimensionality*, which refers to the diminished effectiveness of certain machine learning algorithms in high-dimensional spaces [113]. This work considers two approaches to dimension reduction, singular value decomposition (SVD) and linear discriminant analysis (LDA). SVD is a matrix factorization algorithm, similar to principal component analysis (PCA), that is used to project the features onto a lower dimensional feature space. The feature space is selected such that the resulting features are uncorrelated. LDA also projects the features onto a lower dimensional space. For LDA, the feature space is selected to maximize the distance between training instances of different classes. Both SVD and LDA require training to determine the appropriate feature projections.

4.4.1 Truncated singular value decomposition (SVD)

Truncated SVD is a matrix factorization used here to project feature vectors $\mathbf{x}$ onto a lower dimensional space [114]. In the context of document classification when used with word frequencies, it is also referred to as latent semantic analysis (LSA). For training, SVD considers the matrix X, formed by horizontally concatenating all of the feature vectors $\mathbf{x}_i$ in the training set $\mathcal{X}$,

$$X = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_{N-1} & \mathbf{x}_N \end{bmatrix}, \quad \text{where} \quad \mathbf{x}_i \in \mathcal{X}\,. \tag{4.7}$$

The rank of the matrix X (*i.e.*, the maximum number of linearly independent feature vectors that can be found in X) is r. The truncated SVD algorithm uses matrix factorization techniques to find an approximation X_k of X. The subscript k indicates that X_k is the rank k approximation of X, where $k \leq r$. The truncated SVD of rank k of X is

$$X_k = U_k \Sigma_k V_k^T\,. \tag{4.8}$$

Here, Σ_k is a $k \times k$ diagonal matrix formed from the square roots of the k largest nonzero eigenvalues of XX^T (or equivalently of $X^T X$), U_k is composed of the corresponding orthogonal eigenvectors of XX^T, and V_k is composed of the corresponding orthogonal eigenvectors of $X^T X$. The truncated SVD contrasts with the standard SVD, which considers all r nonzero eigenvalues of XX^T. It is used here instead of the standard SVD to further reduce the number of dimensions. Given the rank k decomposition of X, a feature vector $\mathbf{x}$ can be projected to a k-element feature vector $\mathbf{x}_k$ using the transformation

$$\mathbf{x}_k = \Sigma_k U_k^T \mathbf{x} \tag{4.9}$$

Truncated SVD is effective because the smallest values in Σ_k do not significantly affect the matrix product X_k. Therefore, if only the smallest values are removed, X_k remains a close approximation for X. The value k must be carefully chosen to appropriately reduce redundancy without significantly reducing the discriminability afforded by the original feature set. The value of k is commonly selected experimentally or so that the selected projection retains a fixed percentage of the variance in X.

The motivation for using truncated SVD for feature selection is two-fold. First, it has an advantage over other similar techniques, such as PCA, because it works well with high-dimensional, sparse datasets. Second, when provided with appropriately scaled inputs – *e.g.*, the vectors output by the feature scaling component described in the preceding section – the distribution of the resulting features are approximately Gaussian. Transforming the data to a set of Gaussian-distributed, non-redundant features is especially useful for applying techniques such as LDA, described in the following section.

4.4.2 Linear discriminant analysis (LDA)

LDA is a generalization of Fisher's Linear Discriminant, and is useful for both feature reduction and classification [113]. Whereas SVD transforms the feature data to eliminate redundancy, LDA considers the class labels and transforms the data to separate the instances of different classes. LDA assumes the class-conditional distribution of the features to be Gaussian and requires non-redundant input features. Therefore, LDA is only applied to the SVD-transformed feature vectors $\mathbf{x}_k$. LDA seeks a $k' \times k$ matrix W that projects $\mathbf{x}_k$ onto a space of k' features that maximize the between-class separation of the projected data. The projection is given as

$$\mathbf{x}_{k'} = W\mathbf{x}_k \tag{4.10}$$

where

$$k' \leq \min(k, K-1)\,. \tag{4.11}$$

LDA is not particularly useful for the binary detection task, since a single dimension cannot be reliably used to differentiate between malicious and benign processes. However, it is useful in for classification, where the number of classes – and therefore the number of dimensions of the projected data – is much larger. It is especially useful for classification techniques that work well with smaller feature sets, such as nearest neighbor methods.

During training, the matrix W is selected to maximize the between-class covariance and minimize with within-class covariance of the data in the projected space. It is computed using a training set $\mathcal{X}$ of N traces, wherein the traces are partitioned into K classes, denoted as the sets $\mathcal{C}_k$. The between-class covariance of the projected data, where N_k is the number of training instances that belong to class k, $\boldsymbol{\mu}_k$ is the mean of class k, and $\boldsymbol{\mu}$ is the mean of all the data, is

$$\mathbf{s}_{\mathrm{B}} = \sum_{k=1}^{K} N_k(\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T \,. \tag{4.12}$$

The within-class covariance of the projected data is

$$\mathbf{s}_{\mathrm{W}} = \sum_{k=1}^{K} \sum_{\mathbf{x}_{k'} \in \mathcal{C}_k} (\mathbf{x}_{k'} - \boldsymbol{\mu}_i)(\mathbf{x}_{k'} - \boldsymbol{\mu}_i)^T \,. \tag{4.13}$$

Given these two expressions, the matrix W is selected to maximize the multi-class Fisher criterion $J(W)$, where $\mathrm{Tr}(\cdot)$ indicates the matrix trace operation,

$$J(W) = \mathrm{Tr}(\mathbf{s}_{\mathrm{W}}^{-1}\mathbf{s}_{\mathrm{B}}) \,. \tag{4.14}$$

The Fisher criterion varies inversely with the within-class covariance and proportionally with the between class covariance. Thus, finding the projection W that maximizes the Fisher criterion determines a feature space wherein data from similar classes are clustered together and separated from those of differing classes.

4.5 Feature extraction evaluation

This chapter presented feature extraction as a four-step process containing information retrieval, feature selection, feature scaling, and feature reduction steps. With the exception of information retrieval, all of these steps include a training component, wherein labeled training data are used to determine the parameters of the transformation. In practice, these parameters are learned using the same set of training data used to train the malware detectors and classifiers described in the following chapters. The parameters learned during training are used to transform the data during testing.

One of the contributions of this thesis is the analysis of the effects feature extraction has on malware detection and classification accuracy (Sections 5.4.2 and 6.3.2). Combinations of the described techniques and parameters are used for feature extraction, and their effectiveness is characterized by the accuracy of the detectors and classifiers with which they are used. The combination techniques and parameters that afford the highest detection and classification accuracy are chosen for use in the overall malware detection and classification system.

Chapter 5: Malicious Process Detection

This chapter addresses the second major challenge of this thesis, the detection of malicious processes. The three primary challenges in malicious process detection are

- differentiating between malicious and benign processes on production hosts in real-time;
- detecting malicious processes as quickly as possible, to mitigate adverse effects of executing malware; and
- achieving a low enough false positive rate (FPR) to facilitate production deployment of the detection system.

Section 5.1 addresses the first challenge by identifying machine learning and signature-based techniques that are 'lightweight' enough to use for online detection. The chosen techniques were selected for their low computational complexity during detection and their previously demonstrated effectiveness in both document classification and malware detection. Section 5.2 addresses the second challenge by providing sequential detection techniques that can be used for rapid malware detection. The remaining sections address the third challenge by providing experimental evaluation and comparison of the detection algorithms described in this chapter and the feature extraction techniques described in Chapter 4. Section 5.3 describes detector evaluation techniques, Section 5.4 presents the experimental results, and Section 5.5 presents a case study wherein the described detection system was used to detect malicious processes in public computer laboratories.

5.1 Malware detection algorithms

This study considers four different techniques for malicious process detection:

- a signature-based detector (Section 5.1.1),
- a multinomial log-likelihood ratio test (LLRT) (Section 5.1.2),

- support vector machines (SVMs) (Section 5.1.3), and
- logistic regression (LR) (Section 5.1.4).

Each technique is used in a supervised learning context, wherein the models used for detection are learned from labeled training data. The labeled training data consist of system call traces collected from known malicious and benign processes. Each technique includes a training algorithm and a testing algorithm. The training algorithm is used for model creation and the testing algorithm is used for model evaluation.

The four techniques offer different approaches to detection. The signature-based detector identifies potential malware by comparing observed system call n-grams to a set of known malicious n-grams. The log-likelihood ratio test (LLRT) models the class-conditional probability distributions of the training data, and uses the models to compute the probability that the feature data come from a malicious process. Finally, the support vector machine (SVM) and logistic regression (LR) are linear detectors that represent the training data in Euclidean space. SVMs seek a hyperplane to linearly separate the data, and LR develops a model to compute the probability a process is malicious.

5.1.1 Signature-based detector

Signature-based detection techniques are especially popular in the related work in intrusion detection, where anomalous system call sequences are used to identify potential intrusions (Section 2.1.3). Signature-based techniques are popular because of the white-box models they provide. That is, a positive detection can be directly attributed to specific events. For example, when a malware sample is detected using the signature-based detector described in this work, the specific n-grams that caused the positive detection are easily identified. This study considers a signature-based detector that identifies potential malware by comparing the n-grams in a system call trace to a set of malware signatures, *i.e.*, a set of n-grams previously seen only in known malware. Unlike the other detectors presented in this chapter, the signature-based detector uses only information retrieval for feature extraction. n-gram frequency vectors.

The signatures are created by considering a training set $\mathcal{X}$ of feature vectors $\hat{\mathbf{x}}$ extracted from known malicious and benign processes. The vectors are partitioned into two sets, those collected from benign processes $\mathcal{X}_{\text{B}}$, and those collected from malicious processes $\mathcal{X}_{\text{M}}$. The signatures are created by identifying the set of features present in $\mathcal{X}_{\text{M}}$ but not in $\mathcal{X}_{\text{B}}$. The signatures are represented by a binary vector $\mathbf{s}$ that indicates which of the n-grams are members of the signature set. The vector $\mathbf{s}$ is defined as the following, where s_i and $\hat{x}_i$ are the i^{th} components of the signature vector $\mathbf{s}$ and feature vector $\hat{\mathbf{x}}$,

$$\mathbf{s}_i = \begin{cases} 1, & \text{if } \left(\sum_{\hat{x} \in \mathcal{X}_{\text{M}}} \hat{x}_i \right) > 0 \text{ and } \left(\sum_{\hat{x} \in \mathcal{X}_{\text{B}}} \hat{x}_i \right) = 0, \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

For detection, the signature detector computes the number of system calls in a trace that match signatures of known malware, and compares the result to a threshold λ. The detector labels a process with a feature vector $\hat{\mathbf{x}}$ malicious if

$$\mathbf{s}^T \hat{\mathbf{x}} > \lambda\,. \tag{5.2}$$

The threshold λ is chosen experimentally to fine-tune the detector performance. While the advantages of the signature-based detector are the simplicity of its training and testing algorithms and its white-box model, its weakness is its simplistic characterization of malicious behavior that does not generalize well to new malware samples and its tendency to overfit the training data.

5.1.2 Multinomial log-likelihood ratio test (LLRT)

The naïve multinomial LLRT is a statistical test used in document classification [115]. It operates on discrete-valued, non-negative feature data, such as the n-gram frequencies and term frequency – inverse document frequency (TF-IDF) transformed feature data considered in this study. It models the class-conditional distributions of the feature data using a categorical distribution model, assuming conditional independence of the features. It is desirable due to the simplicity of its training and detection algorithms and can provide useful detection results even when the independence assumption does not hold.

During training, the algorithm estimates the class-conditional probability distributions of each feature $\hat{x}_i$ from the training data. In this study, the probability distribution of the i^{th} feature is described as the probability that the random variable (r.v.) $\hat{X}_i$ corresponding to the i^{th} feature takes on the value $\hat{x}_i$. The class-conditional distributions of the training data are represented as $p_{\hat{X}_i|M}(\hat{x}_i)$ for the malicious processes and $p_{\hat{X}_i|B}(\hat{x}_i)$ for the benign processes. For each possible value $\hat{x}_i$, the probability density function $p_{\hat{X}_i|M}(\hat{x}_i)$ is estimated from the training data as the fraction of total malware training instances for which the r.v. $\hat{X}_i$ takes on the value $\hat{x}_i$. That is, given $N_{\text{M}} = \textbf{card}(\mathcal{X}_{\text{M}})$,

$$p_{\hat{X}_i|M}(\hat{x}_i) = \frac{1}{N_{\text{M}}} \sum_{\hat{x} \in \mathcal{X}_{\text{M}}} \mathbf{1}_{\hat{X}_i = \hat{x}_i} \tag{5.3}$$

The probabilities $p_{x_i|B}(x_i)$ are similarly computed from the benign training data. Given a feature vector $\hat{\mathbf{x}}$, detection is performed by computing the likelihood ratio

$$\Lambda = \frac{p_{\hat{\mathbf{X}}|\text{M}}(\hat{\mathbf{x}})}{p_{\hat{\mathbf{X}}|\text{B}}(\hat{\mathbf{x}})} \tag{5.4}$$

The conditional independence assumption enables the probabilities $p_{\hat{\mathbf{X}}|\text{M}}(\hat{\mathbf{x}})$ and $p_{\hat{\mathbf{X}}|\text{B}}(\mathbf{x})$ to be represented as the product of the marginal probabilities $p_{\hat{X}_i|M}(\hat{x}_i)$ and $p_{\hat{X}_i|B}(\hat{x}_i)$, respectively. Given m as the total number of features, the detector performs a log-likelihood ratio test, wherein a process is labeled malicious if

$$\sum_{i=1}^{m} \log \left(\frac{p_{\hat{X}_i|\text{M}}(\hat{x}_i)}{p_{\hat{X}_i|\text{B}}(\hat{x}_i)} \right) > \lambda. \tag{5.5}$$

The detection threshold λ can be chosen under the Bayes criterion, based on error costs and prior probabilities; or under the Neyman-Pearson criterion, by maximizing the true positive rate (TPR) under a FPR constraint [116]. This work uses the latter approach, since prior probabilities that a process is benign or malicious are in general not known.

The multinomial LLRT detector works with the TF-IDF transformed n-gram frequencies because they are positive and discrete-valued. However, it does not work with feature extraction techniques such as singular value decomposition (SVD) or linear discriminant analysis (LDA), which result in feature data that can be approximated with a Gaussian distribution model. In such cases, the LLRT can be used with Guassian marginal distributions. Here, the class-conditional marginal distribution of i^{th} feature for the malicious processes is

$$p_{\hat{X}_i|\mathrm{M}}(\hat{x}_i) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left(\frac{-(\hat{x}_i - \mu_i)^2}{2\sigma_i^2}\right) , \tag{5.6}$$

where μ_i is the maximum likelihood estimator of the mean of the training data,

$$\mu_i = \frac{1}{N_\mathrm{M}} \sum_{\mathcal{X}} \hat{x}_i \, , \tag{5.7}$$

and σ_i^2 is the the maximum likelihood estimator of the standard deviation of the training data,

$$\sigma_i^2 = \frac{1}{N_\mathrm{M}} \sum_{\mathcal{X}} (\hat{x}_i - \mu_i)^2 \ . \tag{5.8}$$

The marginal probabilities $p_{x_i|\mathrm{B}}(x_i)$ are similarly computed from the benign training data.

The LLRT detector is advantageous because its output probabilities can be used with sequential detection [117] or data fusion techniques [118] that require probabilities of observations as input. However, it is limited by its independence assumption, assumed marginal probability distributions, and tendency to overfit the training data.

5.1.3 Linear support vector machines (SVMs)

Linear SVMs seek a hyperplane $\mathbf{w}^T\hat{\mathbf{x}}$ that optimally separates data points of two different classes. The parameters $\mathbf{w}$ defining the hyperplane are learned from labeled training data. Detection is performed on a feature vector $\hat{\mathbf{x}}$ by comparing the weighted sum $\mathbf{w}^T\hat{\mathbf{x}}$ to a threshold. A process is labeled malicious if

$$\mathbf{w}^T\hat{\mathbf{x}} > \lambda \, . \tag{5.9}$$

In this work, the parameters $\mathbf{w}$ are calculated using stochastic gradient descent (SGD). SGD is an optimization algorithm well suited to large-scale learning problems such as this, where both the number of training instances and the number of features is very large, and the data are sparse [119]. The objective function SGD seeks to minimize is given in terms of the feature vectors $\hat{\mathbf{x}}_i$ and their corresponding labels $y_i \in \{-1, 1\}$, where $y_i = 1$ for malware traces and $y_i = -1$ for benign software traces. The objective function considers a regularization constant α, and a loss function L. The regularization constant penalizes high model complexity, and the loss function penalizes detection errors during training. Soft margin SVMs are used in this work, which allow for detection errors during training. The advantage of using soft margin SVMs is that they allow for mislabeled training data and are less sensitive to outliers than the hard-margin implementation. For soft margin SVMs, the loss function L is the Hinge Loss,

$$L(t, y) = \max(0, 1 - ty)\,, \tag{5.10}$$

and objective function is

$$E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \mathbf{w}^T\hat{\mathbf{x}}_i) + \alpha\|\mathbf{w}\|_2\,. \tag{5.11}$$

SGD approximates the gradient of $E(\mathbf{w})$ by randomly iterating over the training data $\mathcal{X}$ and considering each training vector $\mathbf{x}$ individually. The training algorithm makes a fixed number of passes over the training data to arrive at an estimated solution for $\mathbf{w}$. The complexity of the SGD training algorithm for $n > 1$ is $O(NPl)$, where N is the number of training samples, P is the number of passes, and l is the number of system calls in a trace. For each training sample, the SGD training algorithm updates the weights considering an adaptive learning rate η_t, which determines the contribution of each estimated gradient to the feature weights. In this study, η_t is gradually decaying. Given an initial value t_0 and the current training iteration t, the learning rate is

$$\eta_t = \frac{1}{\alpha(t_0 + t)} \tag{5.12}$$

The weights $\mathbf{w}$ are updated according to the approximate gradient and learning rate as

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t\left(\frac{\partial L(y_i, \mathbf{w}^T\hat{\mathbf{x}}_i)}{\partial \mathbf{w}} + \alpha\frac{\partial(\|\mathbf{w}\|_2)}{\partial \mathbf{w}}\right)\,, \tag{5.13}$$

where $i = t \bmod N$ and the ordering of the training samples in $\mathcal{X}$ is randomized between passes.

The SVM detector is advantageous because it does not operate on the same restrictive assumptions as the LLRT regarding the distribution and independence of the feature data. Instead, the SVM detector assumes that the data are linearly separable. Furthermore, the SGD training algorithm is advantageous because it works well with large numbers of training instances, high dimensional datasets, and sparse datasets. Since it operates on training vectors individually, the SGD algorithm enables the weights $\mathbf{w}$ to be updated as more training instances become available. This is advantageous because the models are expected to require retraining to account for changes in both benign and malicious software.

5.1.4 Logistic regression (LR)

Logistic regression (LR) is closely related to SVM, but instead of seeking a separating hyperplane, LR provides a model for estimating the probability that a process is malicious or benign [120]. LR is desirable because its probability estimates can be used with sequential detection [117] or data fusion techniques [118] that require probabilities of observations as input. The objective function for LR is the same as expression 5.11, using logarithmic loss instead of Hinge loss,

$$L(t, y) = \log\left(1 + \exp\left(-ty\right)\right) . \tag{5.14}$$

The feature weights $\mathbf{w}$ can be used to compute the probability that a process is malicious as

$$p_{\mathrm{M}}(\hat{\mathbf{x}}) = \frac{1}{1 + e^{-(\mathbf{w}^T \hat{\mathbf{x}})}} , \tag{5.15}$$

and the probability that a process is benign as

$$p_{\mathrm{B}}(\hat{\mathbf{x}}) = 1 - p_{\mathrm{M}}(\hat{\mathbf{x}}) . \tag{5.16}$$

A process is labeled malicious if the log-likelihood ratio of these quantities exceeds a threshold,

$$\log\left(\frac{p_{\mathrm{M}}(\hat{\mathbf{x}})}{p_{\mathrm{B}}(\hat{\mathbf{x}})}\right) > \lambda . \tag{5.17}$$

5.2 Sequential malware detection

Sequential detection, also known as sequential hypothesis testing, is the process of performing detection when the sample size is not known in advance. For malware detection, the sample size is l, the length of the system call traces. The detection algorithms described in the preceding sections assume that the trace length is fixed and perform detection when exactly l system calls have been observed. A sequential detector operates as the traces are collected and makes a decision when there is sufficient evidence to support the decision.

The motivation for using sequential detection techniques here is two-fold. First, they enable more rapid detection of malicious processes than is possible with fixed sample size techniques. This is useful to minimize the adverse effects of executing malware. Second, they enable the detection of changes in the behavioral characteristics of a process. This is important for detecting so-called *latent malware*, which only exhibit malicious behaviors when certain conditions are met. Furthermore, sequential detection techniques potentially can be used to detect intrusions, exploitation, code injection, and other malicious behaviors that occur in already-executing processes. This section describes adaptations of two different sequential detection techniques, Wald's sequential probability ratio test (SPRT) and Page's cumulative sum (CUSUM) test, to the malware detection problem. Each sequential detection technique is used together with the LR detector described in the preceding section.

5.2.1 Wald's sequential probability ratio test (SPRT)

The sequential detection problem was formulated by Abraham Wald [117] as a simple binary hypothesis testing problem. Given a sequence of independent, identically distributed (i.i.d.) realizations $\langle z_1, z_2, z_3, ...\rangle$ of a r.v. Z, either hypothesis H_0 or H_1 is true

$$\begin{aligned} H_0 :&\quad \langle z_1, z_2, z_3, ...\rangle \overset{i.i.d.}{\sim} p_0 \\ H_1 :&\quad \langle z_1, z_2, z_3, ...\rangle \overset{i.i.d.}{\sim} p_1 \,. \end{aligned} \tag{5.18}$$

That is, the realizations $\langle z_1, z_2, z_3, ...\rangle$ are either drawn from a probability distribution p_0 or p_1. The goal is to identify which alternative is true as quickly as possible, subject to TPR and FPR constraints. The SPRT uses a sequential probability ratio Λ_i as a decision statistic. The ratio Λ_i is defined recursively, where $\Lambda_0 = 0$, as

$$\Lambda_i = \Lambda_{i-1} \times \frac{p_1(z_i)}{p_0(z_i)} \tag{5.19}$$

Whereas the fixed sample size detection algorithms described previously in this chapter each considered a single decision threshold λ, Wald's SPRT considers two decision thresholds, λ_1 and λ_0. At each iteration, one of the following actions is taken.

- If $\Lambda_i > \lambda_1$, stop and declare H_1 to be true.
- If $\Lambda_i < \lambda_0$, stop and declare H_0 to be true.
- Otherwise, continue.

The thresholds are chosen to achieve a desired FPR and TPR. Conservative choices for the thresholds are

$$\begin{aligned} \lambda_1 &= \frac{\text{TPR}}{\text{FPR}}\,, \\ \lambda_0 &= \frac{1-\text{TPR}}{1-\text{FPR}}\,. \end{aligned} \tag{5.20}$$

The present problem of sequential malware detection has two primary differences from the sequential detection problem posed by Wald. First, the two detection algorithms that provide probabilities as output (LR and the LLRT) are machine learning algorithms operating on high-dimensional datasets and not simple binary hypothesis tests. Second, each new data sample in the malware detection problem is a new system call n-gram, and not an independent realization of a r.v. The first difference necessitates the experimental determination of the decision thresholds λ_0 and λ_1, since expression 5.20 was derived from the assumption that a simple binary ratio test was being performed. The second issue is addressed by performing block-wise feature extraction on the system call traces, using a fixed block size l_b. For example, a trace containing 1500 calls can be viewed instead as a sequence of 15 disjoint traces of length $l_b = 100$. The complete sequential malware detection procedure using Wald's SPRT is described by the following steps.

1. Collect l_b system calls.
2. Perform feature extraction on the system calls to determine $\hat{\mathbf{x}}_i$
3. Perform detection on $\hat{\mathbf{x}}_i$ using LR.
4. Compute the sequential probability ratio in expression 5.19.
5. Perform the two threshold tests.
6. If no decision has been made, repeat.

The procedure is repeated until one of the two alternatives is chosen, *i.e.,* the process is declared 'benign' or 'malicious'.

5.2.2 Page's cumulative sum (CUSUM) test

Wald's SPRT begins when a process is created and terminates when the process is determined to be benign or malicious. Since the detector makes only a single decision and bases it only off of the beginning of a process's execution, a malicious process could evade detection by initially behaving like a benign process. This type of attack, known as a mimicry attack, could similarly be used against any of the fixed sample size tests described in the previous section, which operate only on the first l calls in each system call trace.

This section considers an alternative sequential test chosen to address the concern that malware might evade detection by delaying their malicious functions. Page's CUSUM test, originally described as a method for detecting faults in industrial processes, formulates the sequential detection problem as a quickest change-point detection problem [121]. The goal in change-point detection is to detect an abrupt change in a process. Whereas the goal in Wald's SPRT is to determine whether a sequence $\langle z_1, z_2, z_3, ...\rangle$ is drawn from a distribution p_0 or p_1, the goal in Page's CUSUM test is to identify if and when the distribution of the realizations z_i change from p_0 to p_1.

Page's CUSUM test uses the same decision statistic as the SPRT, the recursively defined cumulative likelihood ratio defined in expression 5.19. However, instead of setting two fixed thresholds, the CUSUM test uses a single adaptive threshold. The intuition behind Page's CUSUM test is that as long as H_0 is more likely, the ratio

$$\frac{p_1(z_i)}{p_0(z_i)} \tag{5.21}$$

will tend to be less than 1, whereas it will tend to be greater than 1 if H_1 is more likely. The test decides H_1 to be true if a significant increase in Λ_i is observed. At each iteration, the CUSUM test takes one of the two following actions.

- If $\Lambda_i - \min\limits_{j \leq i}(\Lambda_j) > \lambda$, stop and declare H_1 to be true.
- Otherwise, continue.

Page's CUSUM test does not terminate until the hypothesis H_1 is determined to be true. Thus, when used for sequential malware detection, the CUSUM test continues to monitor every process until malicious behavior is detected or until the process terminates. For malware detection, the CUSUM test is used together with the LR detector, similar to the approach taken with Wald's SPRT. The CUSUM test is performed block-wise and the detection threshold λ is determined experimentally.

5.3 Malware detector evaluation

This chapter presented four detection algorithms and two sequential detection procedures for identifying malicious processes. To determine the effectiveness of these detection techniques, each was experimentally evaluated against system call traces collected from production environments and from recently discovered malware samples. This section describes the performance measures and cross-validation techniques used to evaluate the described malware detectors.

5.3.1 Detector performance measures

Detection performance is studied primarily in terms of two values,

False positive rate (FPR): the fraction of benign processes incorrectly identified as malware; and

True positive rate (TPR): the fraction of malware samples correctly identified as malware.

While the FPR is defined in terms of the processes, the TPR is defined in terms of malware samples. This distinction is made because the number of processes created by the malware samples in this study varied significantly, with some samples creating thousands of processes and others creating only one. Presenting the TPR in terms of malware samples prevents malware samples composed of many processes from biasing the results.

The receiver operating characteristic (ROC) curve of a detector is a plot of the detector's TPR versus its FPR, created by varying the detection threshold λ and plotting the TPR and FPR achieved at each threshold. The ROC curve depicts the trade-off between the TPR and the FPR of a detector. The area under the curve (AUC) of the ROC can be used to quantify the overall accuracy of a detector. A perfect detector has an AUC of 1, whereas a detector that makes decisions randomly has an AUC of 0.5.

This study focuses primarily on detector performance under specific FPR and TPR constraints, as indicated by Neyman-Pearson lemma [116]. In particular, this study presents the maximum TPR achieved by a detector at a fixed FPR. This study focuses on the maximum TPR that a detector can achieve at a FPR below 10^{-5}. For the data used in this study, a FPR of 10^{-5} corresponds to approximately one false positive every 17.5 host-days.

The sequential detection algorithms are evaluated in in terms of their FPR, TPR, and average detection delay (ADD), which can be tuned by adjusting the detection thresholds. Here, the ADD is the average delay between the start of a process and a decision being made by the detector. The ADD is measured as the average number of system calls observed by the detector before a decision is made.

5.3.2 Cross-validation

Cross-validation is a model validation technique used to assess the effectiveness of the detection techniques described in this chapter and the classification techniques presented in the next chapter. Cross-validation is the process of partitioning a dataset into complementary subsets, using one subset for training and the other for testing. Cross-validation is useful for assessing how a detector will perform against an independent dataset.

In this work, cross-validation is performed by splitting the data into training and testing sets along dataset boundaries. That is, the training-testing splits never occur within a set of traces collected during the same System Call Service (SCS) data collection session. This partitioning scheme ensures that the set of malware samples represented in the training and testing sets are always disjoint, *i.e.*, detection accuracy is only evaluated against a different set of malware samples than are is for training.

Ten-fold cross-validation was used to obtain the results presented in this thesis. In 10-fold cross validation, the data are partitioned into ten subsets. At each cross-validation fold, one subset is chosen as the testing set, and the remaining nine are used for training. The results of 10-fold cross-validation are presented in terms of the mean performance achieved across the cross-validation folds. The measurements of the performance of a detector across 10 folds are represented as $\mathcal{Z} = \{z_1, z_2, ..., z_{10}\}$. In the following section, the average performance is presented along with its standard error, the standard error of the mean (SEM), where $\sigma_{\mathcal{Z}}$ is the variance of $\mathcal{Z}$,

$$\text{SEM}(\mathcal{Z}) = \frac{\sigma_{\mathcal{Z}}}{\sqrt{10}}\,. \tag{5.22}$$

In discussions of significance in the analysis of the detection results, an independent two-sample t-test is used to assess whether differences in performance are statistically significant. The two-sample t-test computes the t statistic from two sets of measurements, $\mathcal{Z}_1$ and $\mathcal{Z}_2$ as the following, where $\mu_{\mathcal{Z}_i}$ and $\sigma^2_{\mathcal{Z}_i}$ are the mean variance of the i^{th} set,

$$t = \frac{\mu_{\mathcal{Z}_1} - \mu_{\mathcal{Z}_2}}{\sqrt{\frac{1}{2}\left(\sigma^2_{\mathcal{Z}_1} + \sigma^2_{\mathcal{Z}_2}\right)}\sqrt{\frac{2}{10}}}\,. \tag{5.23}$$

In this thesis, results are evaluated to a 5% significance level, *i.e.*, a difference is considered statistically significant when the p-value of the t-test is below 5%.

5.4 Experimental malware detection results

The detection results presented in this section are presented primarily in terms of the TPR achieved by the detectors at a fixed FPR of 10^{-5}. The results were achieved through cross-validation performed on the data collected from the home and campus environments, and from the malware testbed. Among the experimental results described in this section are

- a comparison of the four detection algorithms (Section 5.4.1),
- a comparison of the feature extraction techniques (Section 5.4.2),
- a detailed analysis of the most informative n-grams (Section 5.4.3),
- a study of the effects of drift on detection (Section 5.4.6), and
- a study of the accuracy of the detector against specific malware families (Section 5.4.7).

The experimental results presented in this section are used to identify the algorithms and parameters that provide the highest detection accuracy.

5.4.1 Detector, n-gram length, and trace length comparison

This section explores of the effects of three parameters on detection accuracy,

- the chosen detection algorithm,
- the length n of the system call n-grams, and
- the length l of the system call traces.

This study compares the performance of each of the four detection algorithms described in Section 5.1, the LR, SVM, signature-based (SIG), and multinomial LLRT detection algorithms. With the exception of the signature-based detector, the following feature extraction strategy was used:

- ordered n-grams of lengths $n \in \{1, 2, 3, 4, 5\}$;
- system call traces of lengths $l \in [100, 1500]$;
- logarithmic n-gram frequencies;
- inverse document frequency (IDF) transformation; and
- L_2 normalization (for the SVM and LR detectors only).

For the signature-based detector, none of the feature transformations were applied.

The above feature selection strategy was not chosen arbitrarily, but rather selected through an iterative experimental process used to determine the best combination of feature selection parameters. However, since the selection of feature extraction strategies, trace lengths, n-gram lengths, and detectors are inherently coupled, the experimental results of this process are presented here to narrow the scope of the results in the following section to a single detector and n-gram length.

Figure 5.1 provides the experimental results of this analysis. The figure includes five plots, one for each value of $n \in \{1, 2, 3, 4, 5\}$. Each plot shows four traces, one for each of the detectors. The traces show the TPR of the detector for a fixed FPR of 10^{-5} versus the trace length l. The four primary conclusions drawn from the results shown in Figure 5.1 are summarized below.

First, system call frequencies alone are insufficient for the detection of malicious processes. The first plot in Figure 5.1 shows the detection results when $n = 1$. The 1-gram representation of a system call trace describes the frequency at which each system call appears in a trace. The plot shows the signature-based 1-gram detector achieved a TPR of 0 at every trace length. Furthermore, the ROC curve of the signature-based detector (not shown) had an AUC of 0.5, indicating that it provided no discrimination at all, even at higher FPRs. This result indicates that there are no system calls that are used exclusively by malware. The other three detectors do provide some

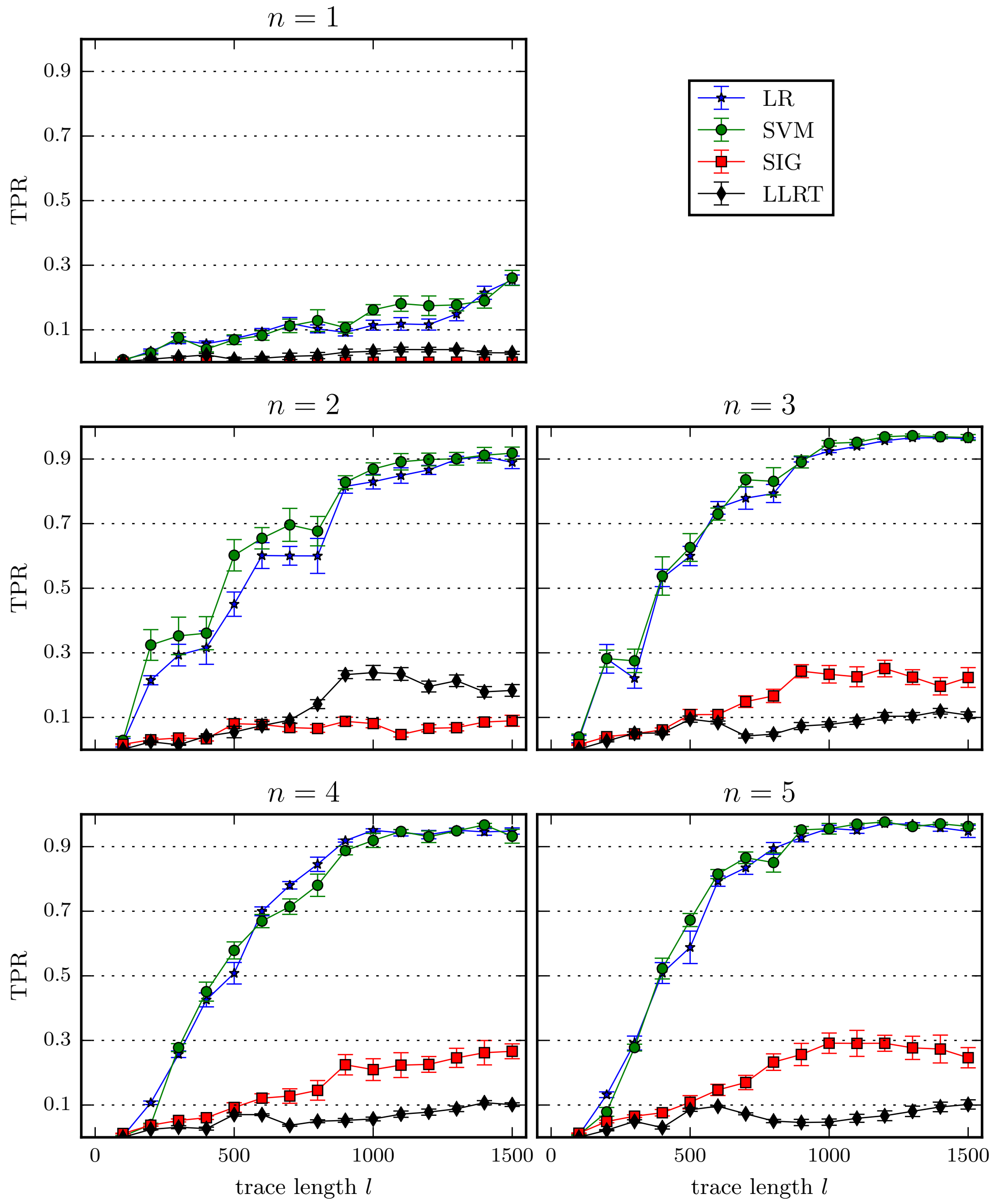

Figure 5.1: Comparison of the LR, SVM, SIG, and LLRT detectors, comparing performance for different n-gram lengths and system call trace lengths l, with detector TPRs presented at a fixed FPR of 10^{-5}

discrimination, shown in Figure 5.1 by the nonzero TPR they achieved at higher trace lengths. The maximum TPR achieved by a 1-gram detector did not exceed 30%, whereas considering any value of $n > 1$ provided a TPR that exceeded 90%. Thus, system call n-gram analysis offers much higher detection accuracy than can be achieved using system call frequencies alone.

The LLRT and signature detectors perform poorly at low FPRs. For $n \in \{1, 2, 3, 4, 5\}$, both detectors significantly underperformed the LR and SVM detectors. The main drawbacks that limit the effectiveness of the LLRT detector are the multinomial probability models it uses for detection and its assumption of conditional independence of the features. The multinomial probability models overfit the training data and do not generalize well to the testing data. Furthermore, conditional independence is not a good assumption for the system call n-gram data. The inclusion of highly correlated features and uninformative features contributes to its poor detection performance. The signature-based detector is also limited by the simplicity of its model. The n-gram signatures identified during training are mainly artifacts of the individual malware samples and do not generalize well to detecting other malware samples.

Detection accuracy peaks at a trace length of around 1000 system calls. This effect is most pronounced for the performance of the LR and SVM detectors for $n \in \{3, 4, 5\}$. While detection accuracy generally increases with trace length for $l < 1000$, increasing l beyond 1000 did not offer statistically significant increases in detection accuracy. The detector was able to identify the studied malware samples based on their behavior during their initial phases of execution. The observed behaviors of the malware samples during their initial execution included

- propagation within a host or over the network;
- 'phoning-home' to indicate to a remote server when a new host was compromised;
- modifying the host configurations to hide the presence of malware;
- installing the malware to ensure their permanence;

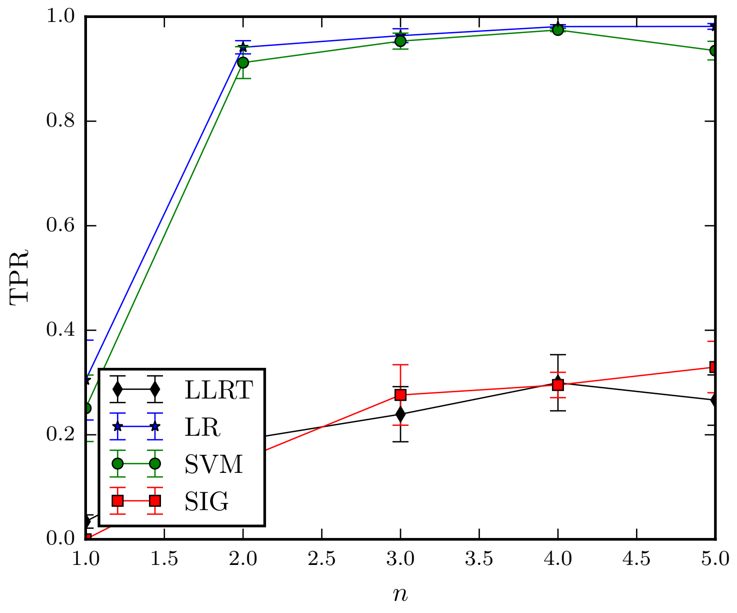

Figure 5.2: Detector n gram length analysis, showing the maximum TPR observed at FPR = 10^{-5} for n-gram lengths $n \in \{1, 2, 3, 4, 5\}$

- disabling security software and other functions that might be used for malware removal or detection;
- spawning additional malware processes;
- collecting information about infected hosts;
- activating keyloggers, password stealer (PWS), and other spyware components; and
- installing additional malware samples on infected hosts.

Maximum detection accuracy is achieved when $n = 3$. Figure 5.2 shows the maximum TPR achieved by each detector at each n value in Figure 5.1. For each value of n, it shows the maximum TPR achieved at a fixed FPR of 10^{-5}. The figure shows that using $n > 3$ does not provide statistically significant improvement in the TPR for the LR or SVM detectors. When $n > 3$, both the dimensionality of the data and the number of nonzero features exceed the number of processes included in the training set. This likely contributes to the overfitting of the detectors to uninformative features.

The detection results presented in this section serve to scope the remainder of the analysis presented in this chapter. First, only the LR and SVM detectors are considered for the remainder of this chapter, since these detectors significantly outperformed the LLRT and signature-based detectors. Since the LR and SVM provided similar results, the remainder of this chapter focuses primarily of the LR detector. Second, only 3-grams of system calls are considered for the remainder of this chapter, since values of $n > 3$ did not offer statistically significant improvement in detector TPR, but did introduce substantial memory and processing overhead.

5.4.2 Feature extraction comparison

As illustrated in the previous section, feature extraction strategy has a significant effect on overall detection performance. While the previous section focused on the effects of n-gram and trace length, this section explores the effects of four additional feature extraction options. These options are

- using (`1`) L_1 or (`2`) L_2 normalization for feature scaling,
- using (`F`) raw frequencies or (`L`) logarithmic-frequencies for term frequency (TF) computation,
- using (`T`) TF transformation alone or (`I`) TF-IDF transformation, and
- using (`O`) ordered or (`U`) unordered 3-grams.

Each of the four techniques is uniquely identified by the single character labels provided in parenthesis in the list above. In this section, all sixteen possible combinations of these options are compared for the LR detector, using 3-grams of system calls and a system call trace length of $l = 1500$. Figure 5.3 shows the TPR achieved at a fixed FPR of 10^{-5} for each of the sixteen combinations. The conclusions drawn from these results are summarized below.

L_2 normalization (`2`) outperforms L_1 normalization (`1`) by a large margin. L_1 normalization was considered because of its straightforward interpretation as the relative frequency of each 3-gram. That is, each feature value represents a fraction of the total 3-grams observed in the trace. However, such features perform poorly in practice mainly due to the LR training and

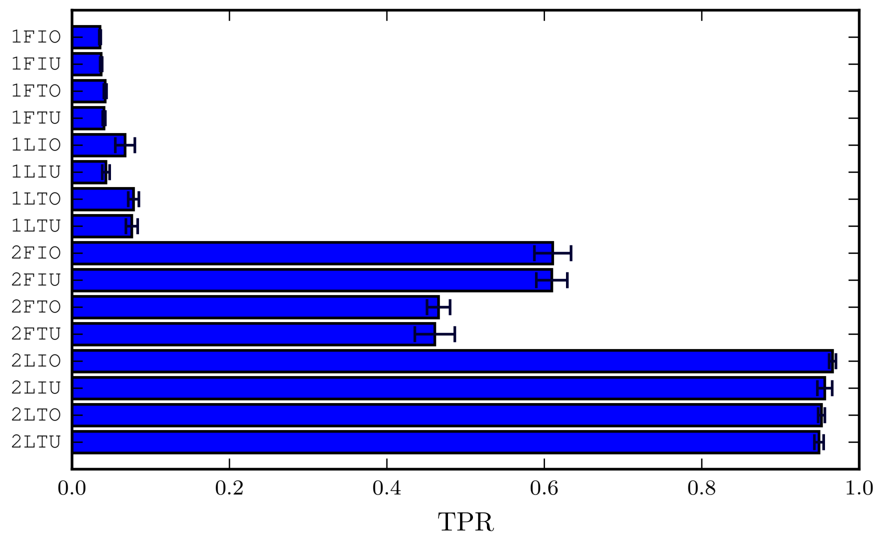

Figure 5.3: Detector feature extraction comparison, showing TPR achieved at FPR=10^{-5} for the feature extraction strategies described in Section 5.4.2

testing algorithms, which perform better with L_2 normalized features. These results are also consistent with results from document classification, which indicate that the L_2 norm provides superior performance [122].

When the L_2 norm (`2`) is used, Logarithmic frequencies (`L`) outperform raw frequencies (`F`) by a large margin. As with words in text documents, the system call n-gram frequencies in this study scale to different orders of magnitude [122]. This problem of differing feature scales is the reason for the comparatively poor performance of the raw frequencies. Although IDF transformation is used to provide higher weights to rare n-grams, the effect of this transformation alone is not significant enough to fully overcome the scaling issue. Using logarithmic frequency is a necessary step to address the issue of feature scale.

The combination of using the L_2 norm, logarithmic frequencies, IDF transformation and ordered n-grams (`2LIO`) provided the highest detection accuracy. The last four entries in Figure 5.3 provide insight into the effects of IDF transformation and system call ordering. The results show that ordered n-grams and IDF transformations each outperform their counter-

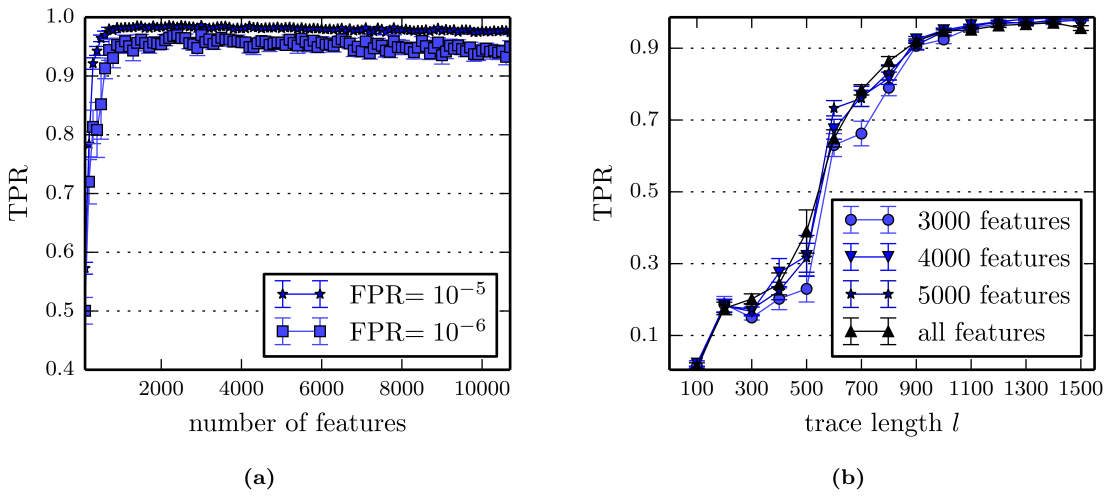

Figure 5.4: Feature selection analysis, showing (a) detector TPR vs. number of selected features and (b) a comparison of three different feature subsets against the set of all features vs. trace length

parts. However, neither of these parameters in isolation offer statistically significant improvement. Using both ordered n-grams and IDF transformation does offer significant improvement over using unordered n-grams and only TF transformation. Thus, the remaining detection results presented in this chapter all use the `2LIO` feature extraction strategy with 3-grams of system calls.

5.4.3 Feature selection

This section focuses on feature selection, the process of identifying the subset of ordered 3-grams that provide the best overall detection performance. Feature selection is used to mitigate overfitting and to decrease the overall memory and computational burden of the detection system. Of the more than 100 million possible ordered 3-grams of system calls, more than 250 thousand distinct 3-grams were present in the system call traces used for feature selection. Feature selection was performed using recursive feature elimination (RFE) and the LR detector, removing 100 features at each recursive step.

Figure 5.4a shows the detector TPR at a fixed FPR of 10^{-5} and 10^{-6}, versus the number of features chosen using RFE. The TPR increases sharply as the number of features increases to around 1,000. After then, there is no statistically significant change in the TPR achieved by considering

Table 5.1: System calls present in the selected feature set of 4,000 3-grams

Category	Calls	Examples
Atoms	4	NtAddAtomEx, NtFindAtom
Boot	2	NtModifyBootEntry, NtAddBootEntry
Debuging	8	NtSystemDebugControl, NtDebugActiveProcess
Device driver control	1	NtUnloadDriver
Environment settings	9	NtSetSystemInformation, NtSetDefaultLocale
Error handling	3	NtRaiseHardError, NtRaiseException
Files and general I/O	29	NtReplacePartitionUnit, NtLockFile
Jobs	4	NtTerminateJobObject, NtAssignProcessToJobObject
local procedure call (LPC)	16	NtCreatePort, NtSecureConnectPort
Memory	17	NtWriteVirtualMemory, NtQueryVirtualMemory
Miscellaneous	36	NtQuerySystemInformation, NtQueryLicenseValue
Objects	12	NtQueryObject, NtSetInformationObject
Plug and play	4	NtPlugPlayControl, NtPlugPlayGetDeviceProperty
Power management	3	NtSetSystemPowerState, NtIsSystemResumeAutomatic
Processes and threads	31	NtQueryInformationProcess, NtOpenProcess
Processor Information	1	NtSetLdtEntries
Registry	24	NtSetValueKey, NtNotifyChangeKey
Security	23	NtImpersonateThread, NtSetSecurityObject
Synchronization	18	NtCreateMutant, NtCreateSemaphore
Timers	8	NtQueryPerformanceCounter, NtCreateTimer

additional features. The maximum average TPR in Figure 5.4a occurs at 3,000 features. The detection performance achieved using the same set of 3,000 features is presented in Figure 5.4b for system call trace lengths $l \in [100, 1500]$. The figure shows the TPR of the detector at a fixed FPR of 10^{-5} versus the trace length l. It compares the performance of the detectors using sets of 3,000, 4,000, and 5,000 features to the performance of the detector using all of the features. The 3,000-feature detector significantly underperforms detector using all the features for certain system call trace lengths, including $l = 500$ and $l = 700$. However, the detector using 4,000 features achieves comparable detection performance to the detector using all the features for every considered trace length. The set of 4,000 features was chosen over the set of 3,000 because it provided comparable performance at $l = 1500$ and better performance at shorter trace lengths.

Of the 465 system calls monitored by the SCS, 253 distinct calls were present in the chosen set of 4,000 features. The selected calls are summarized in Table 5.1, which shows the categories of the calls, the number of calls in each category, and example calls from each category. Every one of the system call categories presented in Section 1.3 is represented in the chosen feature set. Similarly, every category is also represented in the set of features that were not chosen. Therefore, no single class or even collection of system call classes provide the discrimination between malware and benign software.

The chosen 3-grams also illustrate the range of functionality covered by the selected features. Some of the selected 3-grams contained system calls from multiple categories, such as

- ⟨ NtOpenKeyEx, NtSetInformationProcess, NtTraceControl ⟩;
- ⟨ NtRegisterProtocolAddressInformation, NtResumeThread, NtSetSystemPowerState ⟩; and
- ⟨ NtTerminateThread,NtRenameKey,NtDeleteWnfStateData ⟩.

Other selected 3-grams were composed of calls only from a single category, such as

- ⟨ NtOpenKeyEx, NtQueryKey, NtEnumerateKey ⟩ (Registry);
- ⟨ NtQueryAttributesFile, NtOpenFile, NtLockFile ⟩ (Files and general I/O);
- ⟨ NtTerminateThread, NtTerminateThread, NtSuspendThread ⟩ (Processes and threads);
- ⟨ NtAccessCheckByType, NtAccessCheckByType, NtAccessCheckByType ⟩ (Security); and
- ⟨ NtReplyWaitReceivePortEx, NtAlpcConnectPort, NtAlpcSendWaitReceivePort ⟩ (LPC).

The results of the feature selection study presented in this section indicate that there are no simple, intuitive rules for selecting informative n-grams, and that feature selection should be performed empirically. Some approaches in the related work have considered only system call features that seem intuitively informative, such as filesystem, registry, and security-related system calls. However, the results presented here indicate that all categories of system calls are informative for malware detection. Finally, there are significant gains in terms of memory and processing overhead in collecting and processing only the most informative n-grams. Feature selection reduced the feature space from over 100 million features to only 4,000 features.

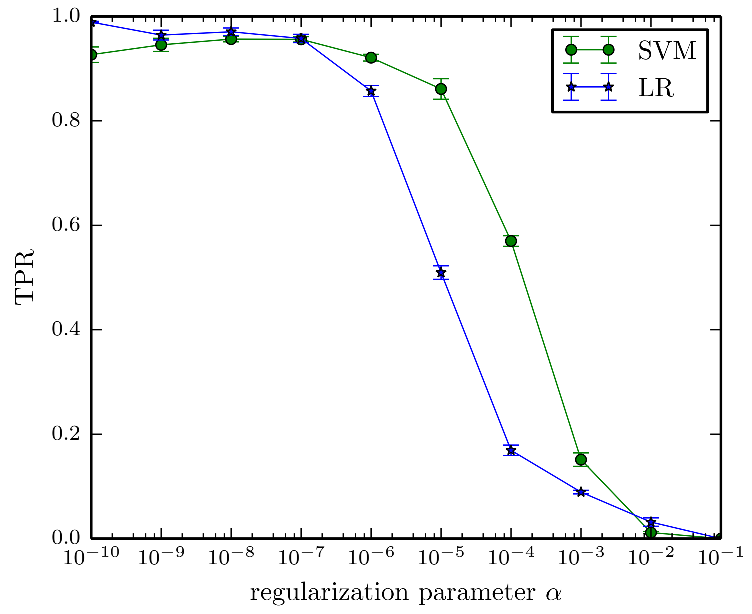

Figure 5.5: Regularization analysis, showing the TPR achieved at FPR=10^{-5} for the SVM and LR detectors for a range of regularization parameters α

5.4.4 Regularization

The optimization problem solved during training for both SVM and LR, shown in expression 5.11 in Section 5.1.4, includes a parameter α, a regularization constant that penalizes high model complexity. The regularization constant penalizes the norm of the feature weights **w** that compose the model. Figure 5.5 presents the accuracy of the LR and SVM detectors for a range of regularization parameters, showing the TPR of the detectors at a fixed FPR of 10^{-5}. In general, the TPR decreases as the regularization parameter α increases, indicating that the detectors benefit from the discriminability afforded by the higher model complexity. Provided α falls within the range $[10^{-9}, 10^{-7}]$, the LR and SVM detectors offer comparable detection performance.

5.4.5 Block-wise detection performance

The detector comparison in Section 5.4.1 indicated that all of the detectors perform relatively poorly for small trace lengths. This observation raises the question: Is the poor performance at small trace lengths an artifact of the relatively small sample size, or is it caused by a lack of discriminatory information at the beginning of the traces? To address this question, Figure 5.6a presents the block-

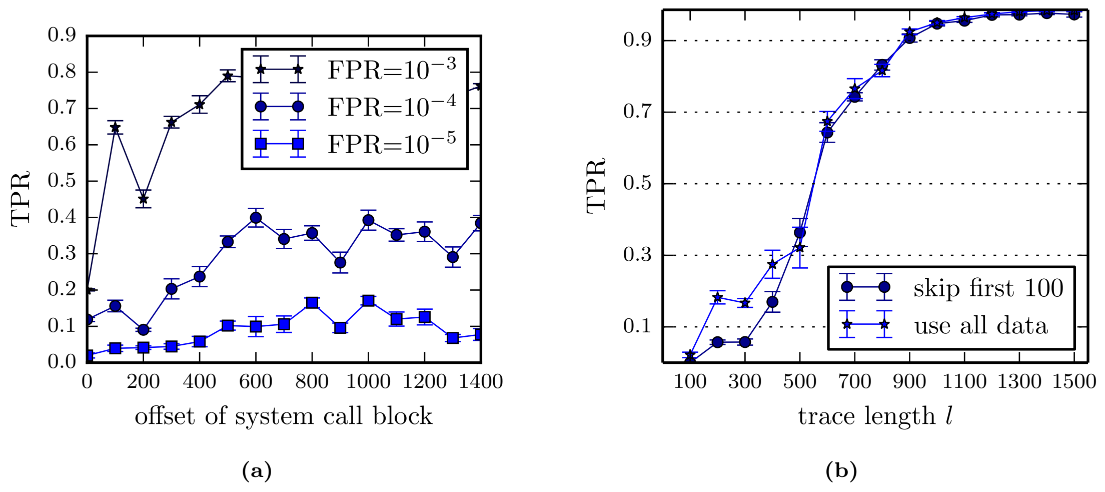

Figure 5.6: (a) Block-wise detection analysis, showing LR TPR for blocks of 100 system calls beginning at the specified offsets and (b) a comparison of the LR detector TPR at a fixed FPR of 10^{-5} when all data are used and when the first 100 calls are ignored

wise detection performance of the LR detector for blocks of 100 system calls. The figure presents the TPR of the detector achieved for each block at fixed FPRs of 10^{-5}, 10^{-4}, and 10^{-3}. Specifically, the first point of the plot shows the detection performance when considering only the first block of 100 system calls from each trace, the second point shows the detection performance for the second block of 100 calls from each trace, and so on.

Figure 5.6a shows comparatively poor performance for the first 100 system calls, especially at an FPR of 10^{-3}. This observation indicates that the first 100 system calls are not as informative as the rest of the calls. This is due in large part to the nature of the calls at the beginning of a processes execution. The calls are mostly related to generic start-up procedures, such as memory allocation, that are common to all processes. For the same reason, the first ~ 500 system calls are also comparatively less informative than the remaining system calls, although this difference is less pronounced.

Figure 5.6b shows the effects of ignoring the first 100 system calls on detection performance. It shows the TPR of the LR detector for a fixed FPR of 10^{-5} versus the trace length. It compares the case where the first 100 system calls are ignored and the case where all of the system calls are considered for detection. Ignoring the first 100 system calls causes a decrease in accuracy for trace lengths less than 500. However, for $l \geq 500$, the detection accuracy matches the accuracy achieved when all the data are used. Therefore, the benefit of ignoring the first 100 calls is only that it slightly reduces the overhead of data collection. It does not provide any benefit in terms of increased detection accuracy.

5.4.6 Effects of drift

Malware authors are continually creating new malware samples and modifying existing samples to add new features, fix bugs, and evade detection. Such incremental changes, referred to as drift, present a challenge for any detector, which must continually adapt to keep up with the changes. Changes introduced specifically to evade detection or make future evasion easier are referred to as adversarial drift [123]. The detection results presented so far in this chapter were achieved using cross validation with randomly determined training-testing splits. While such results provide a measure of how well the detector performs against an independent dataset, these results have no chronological meaning. This section presents a study of detector performance that takes into account the chronological order in which the malware samples were discovered. The objective is to characterize the effects of drift on detection accuracy.

For this study, the dates the malware samples were first submitted to the free online virus scanning service VirusTotal[1] were used as approximate dates of initial discovery. All of the samples were first discovered between January 2012 and March 2015. Cross-validation was performed according to the discovery dates. The detector was trained using all of the malware discovered prior to a chosen date, and testing was performed on the remaining data. Multiple cross-validation folds were performed by sliding the date of the split over the data set. For example, in the first fold, only malware samples discovered before 1 January 2013 were used for training, and the remaining data were used for testing. In the second fold, samples discovered prior to 1 February 2013 were used for training and the rest for testing. This process was repeated fifteen times, and the results were averaged over the folds.

[1]VirusTotal, `http://www.virustotal.com`

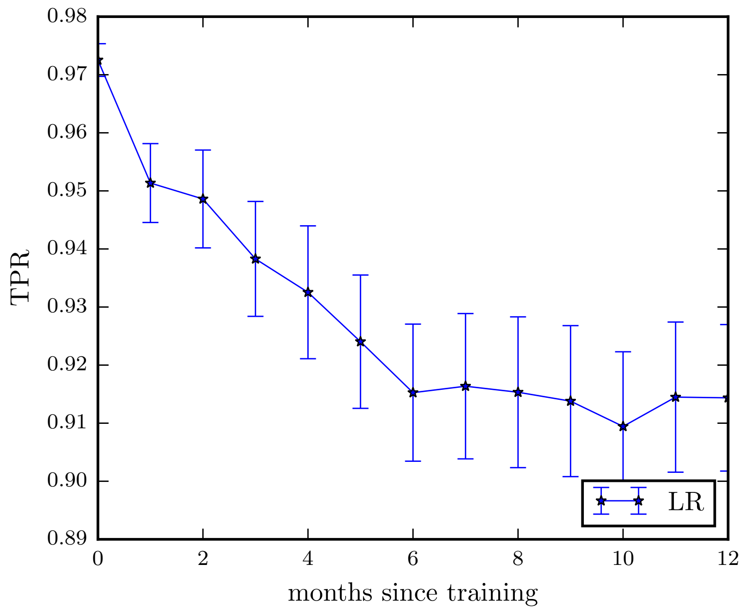

Figure 5.7: Detector TPR at FPR$= 10^{-5}$ as a function of time since training, showing the effects of drift on detection accuracy over a 12 month period

Figure 5.7 shows the TPR of the LR detector at a fixed FPR of 10^{-5} versus the number of months elapsed since training. The results for month zero show the detection performance achieved using cross-validation on the training set. The remaining results are determined by binning the malware according to their month of discovery. The results indicate that detection performance gradually decreases over time. By 12 months, the TPR has decreased by nearly 6.5%. This gradual decrease in detection performance is the effect of drift, caused primarily by the introduction of new malware samples against which the detector performed poorly. The comparatively high standard errors on the measurements compared to month zero are the result of the relatively small sample sizes for the individual months compared to the training set. In fact, most of the months prior to June 2014 – when the malware collection efforts intensified – have fewer than 500 malware samples, while most of the following months have thousands of malware samples.

It is expected that in a production deployment of the detection system, detection models would be updated periodically to correct for drift. The updated models would likely be generated off-line by a third party in a laboratory environment and pushed to clients, similar to the way antivirus

(AV) signatures are currently distributed. The models used for the described detection system have three advantages over traditional malware signatures.

- The models used by the detector are more concise than the large signature databases used in AV software.
- Whereas AV signature databases are typically updated multiple times per day, the models used here would require less frequent updates. For example, if the goal is to stay within 2.5% of peak performance, the models should be updated every three months.
- The models used in this work can be updated using data collected from production hosts, reducing the burden on the vendor of the updates to run every newly discovered malware sample in a laboratory environment.

5.4.7 Error analysis

This section identifies the sources of the detection errors, specifically the types of benign software responsible for the false positives and types of malware responsible for the false negatives. It also examines the hosts on which the detection errors occurred. The results presented in this section are those achieved by the LR detector at a FPR of 10^{-5}.

Seven types of benign software applications were responsible for the false positives, particularly:

- a secure erasure utility,
- a disk defragmentation utility,
- the Windows Defender worker process,
- the Windows narrator,
- two strategy games,
- a component of a graphics editing application, and
- a media player application.

Table 5.2: ESET family labels of false negatives

category	samples
Kryptik	78
Injector	43
AdWare.MultiPlug	36
AdGazelle	28
Softcnapp.C.gen	27
Spy.Banker	24
RiskWare.ShouQu	23
Sality	21
MSIL/Solimba	20
Injector.Autoit	20
Agent	20
AdWare.iBryte	16
Parite	13
Spy.Zbot	12
Sohanad	12
NSIS/TrojanDownloader.Agent	12
MSIL/Kryptik	12
FlyStudio	12
Autoit	12
VB	11

The false positives occurred on six different testbed hosts and one production host. The high representation of the testbed hosts was caused primarily by the fact that they accounted for a majority of the collected data. The common feature among the false positives was that they had relatively low representation in the collected datasets. The cause of the false positives was likely a lack of training data for similar applications, underscoring the importance of training the detector with a variety of benign applications to ensure the models cover a wide range of benign behaviors.

With a TPR rate above 98%, the detector provided 1021 false negatives. The false negatives were studied in terms of their AV labels. According to their Microsoft labels, more than half of the false negatives were not malicious. Those that were labeled malicious by Microsoft spanned 18 malware categories and included 106 Trojans, 69 worms, 69 viruses, and 57 Trojan downloaders. According to their ESET labels, 117 of the false negatives were not malicious. Those that were labeled malicious by ESET spanned 214 different malware families. Table 5.2 presents the twenty malware families that appeared most often in the set of false negatives and the number of occurrences of each. These include multiple adware families, injectors, and a malware family considered *riskware*, *i.e.*, software that is not necessarily malicious. A study of a subset of these samples indicated that they did not perform any obvious malicious tasks on the testbed. The family with the highest representation in the set of false positives was Kryptik, a Trojan backdoor. The reason for its high representation in the set of false positives was primarily its high overall representation, with more than 4,000 samples included in this study.

5.4.8 Malware comparison

The goal of this section is to characterize the LR 3-gram detector's performance against specific malware families. Here, a detector is trained for each malware family and cross-validated against instances of the same family. Table 5.3 summarizes the experimental results achieved for ESET malware families. The table shows the TPR achieved by the detector at a fixed FPR of 10^{-5} against each malware family and the number of samples in each family. Only families with at least 100 samples are shown, and they are sorted in decreasing order of TPR.

The results in Table 5.3 show that the TPR of the detectors were generally high. The detector correctly identified every sample of 23 of the listed families, indicated by the TPRs of 1 in the table. There were also a handful of outliers against which the detector performed relatively poorly. Among those families were LockScreen, ransomware that displays a password-protected lockscreen, and the adware program SystemSecurity. Such false negatives are not particularly worrisome, since these malware samples provide other obvious signs of infection. Among the malware families with the lowest TPRs were malware droppers and downloaders, families that are loosely-defined or that involve the execution of other types of malware. It is likely that such families were not accurately characterized during training, due to their relatively low sample sizes and wide range of functions.

The described malware detection system generally performs very well against well-defined malware families, but struggles against more loosely-defined families. Given this result, it is likely that the use of an ensemble of multiple detectors trained on canonical malware families would lead to an overall improvement in detection accuracy compared to the use of a single malware detector. Such a detector would likely be able to detect the specific processes created by loosely-defined malware samples, such as downloaders and droppers.

Table 5.3: Per-family malware detector performance, showing TPR achieved at 10^{-5} FPR every ESET malware family with at least 100 samples

family	TPR	samples	family	TPR	samples
Mydoom	1.00	1,322	MSIL/Bladabindi	0.99	804
Adware.MultiPlug	1.00	487	AutoRun.VB	0.99	448
MSIL/Solimba	1.00	445	FlyStudio	0.99	576
Yuner	1.00	414	Injector	0.98	4,454
Dlhelper.C	1.00	400	Spy.Banker	0.98	560
AdWare.iBryte	1.00	275	IRCBot	0.98	934
Viking	1.00	274	Autoit	0.98	235
LunaStorm	1.00	245	Sality	0.98	1,961
Neshta	1.00	181	MSIL/Injector	0.98	1,467
DownloadAdmin.I	1.00	126	MSIL/Stimilik	0.98	184
Adware.iBryte	1.00	119	Rbot	0.98	127
bmMedia	1.00	110	Kryptik	0.97	4,088
Spy.Qukart	1.00	110	Wapomi	0.97	280
Klez	1.00	106	Packed.Themida	0.97	175
AdWare.MultiPlug	1.00	1,493	Dorkbot	0.97	186
Simda	1.00	576	Agent	0.97	1,189
SoftPulse	1.00	442	Spatet	0.97	245
Alman	1.00	705	PSW.Fareit	0.96	186
Madang	1.00	699	MSIL/Kryptik	0.96	376
Injector.Autoit	1.00	546	unknown.NewHeur_PE	0.96	196
Packed.ASProtect	1.00	291	Chir	0.96	131
TrojanDownloader.Waski	1.00	314	TrojanDownloader.Agent	0.96	525
AdWare.LoadMoney	1.00	200	TrojanDownloader.Small	0.96	267
Hupigon	0.99	891	TrojanDropper.Agent	0.95	154
Ramnit	0.99	931	TrojanDropper.Delf	0.95	121
ServStart	0.99	304	Neurevt	0.95	108
Parite	0.99	639	Adware.SystemSecurity	0.94	110
Delf	0.99	1,701	MSIL/Agent	0.94	147
Fynloski	0.99	1,051	TrojanDownloader.Banload	0.94	143
Farfli	0.99	653	LockScreen	0.93	233
VB	0.99	621	MSIL/TrjanDropper.Agent	0.91	122
FirseriaInstaller	0.99	250	Virut	0.91	260
PSW.OnLineGames	0.99	965	Generik	0.90	174
MSIL/TrjDropper.Binder	0.99	105	TrojanDownloader.Wauchos	0.89	123
Spy.Zbot	0.99	735			

5.5 Case study

The experimental results in the preceding section were obtained using cross-validation against the data collected from the testbed and from the home and office environments. These are controlled environments wherein the processes could be relatively accurately labeled as benign or malicious in the ground truth. Conversely, the public computer labs present an uncontrolled environment, where typical users are novices with limited training in security best practices and are provided privileged access to the hosts. The goal of this case study is to use the models developed from the controlled environments to detect whether any potentially malicious software executed on the public computers.

For the case study, the LR detector was used in a manner intended to mimic potential commercial deployment. The detector was trained using the labeled data collected from the controlled environments, and the detection threshold λ was set based on the detection results achieved using an independent training set from the same environment. To be consistent with the study in the previous sections, λ was selected as the threshold that provided the maximum TPR for a fixed FPR of 10^{-5}. In total, more than 56,000 processes were observed in the public computer labs. Of these processes, the LR detector labeled 373 processes as potentially malicious. Of the processes labeled malicious,

- 90 were confirmed to be malicious software. These included multiple families of known adware programs and browser hijackers, and were observed on three of the hosts.
- 99 were installers for unknown software whose provenance could not be determined.
- 184 appeared to be false positives.

The false positives were composed of many repeated instances of a small number of applications. The majority (114) of these processes were from bundled software applications pre-installed on the computers by the hardware vendor. The others included a document viewer, a media player, tax preparation software, a game used for teaching students how to type, a massively multiplayer online game, security software, a web browser, and system restore software.

Some of the aforementioned false positives can be attributed to deficiencies in the training data. For example, the training set did not include any hardware vendor software or system restore software, and contained few instances of security software and games. Others, such as the web browser false positives, occurred on a host infected with known adware. Therefore, these processes may have been identified as malware due to malicious browser plugins.

Despite the higher than expected FPR in the case study, the study did have positive takeaways. First, the study did correctly reveal the presence of malware on the public computers. Furthermore, multiple malware processes were identified from malware families that were not included in the malware set used for training. Thus, the case study illustrated that the detector is successful in identifying new malware families in an independent environment. Finally, the case study illustrated flaws in the ground truth labeling process and underscored the importance of using a diverse set of benign software for training.

5.6 Sequential malware detection results

This section presents the sequential detection results achieved using adaptations of Wald's SPRT and Page's CUSUM test. The tests use the probability estimates output by the LR detector to sequentially compute a decision statistic. To apply the sequential detection techniques, the system call traces were partitioned block-wise into sub-traces of a fixed length l_b. The traces studied in the previous section, which contained 1,500 system calls each, were partitioned into sub-traces of sizes ranging from 50 to 750 system calls. Both training and testing were performed block-wise on the sub-traces, and the decision thresholds were determined experimentally to achieve a desired FPR.

The experimental results of this study are presented in Table 5.4, which shows the results achieved at a fixed FPR of 10^{-5} for each sub-trace length l_{b}. The results are presented in terms of the TPR and ADD of each detector. The ADD is presented as the ADD for positive detections only, *i.e.*, the number of system calls on average that were observed before the detector indicated that a process was malicious. This distinction is made to ensure fair comparisons between the two detectors, because the implementation of Page's CUSUM considers only the problem of identifying malware, and not of identifying benign software. The ADD of Wald's SPRT for negative decisions was much lower

Table 5.4: Sequential detection results for Page's CUSUM test and Wald's SPRT, showing the average detection delay (ADD) and true positive rate (TPR) at a fixed FPR of 10^{-5}

l_b	SPRT TPR	SPRT ADD	CUSUM TPR	CUSUM ADD
50	0.81	875	0.72	947
75	0.71	927	0.71	993
100	0.87	881	0.81	949
125	0.86	989	0.81	971
150	0.90	863	0.87	930
200	0.88	874	0.86	907
250	0.91	933	0.87	933
300	0.90	974	0.88	1013
350	0.90	923	0.91	924
375	0.94	912	0.92	960
500	0.94	1030	0.94	1033
750	0.95	1031	0.94	1094

than for positive decisions.

The ADD of the sequential tests was 953 system calls, consistent with the results presented in Section 5.4.1 that indicated peak detection performance was achieved at around 1000 system calls. For the system call traces used in this study, 1,000 system calls corresponds to an average execution time of 205 ms. The ADD of each test was relatively consistent regardless of the choice of the block size l_b. At a fixed FPR of 10^{-5}, both Page's CUSUM test and Wald's SPRT provided similar TPRs. The highest detection accuracy was achieved for block lengths $l_b \geq 500$. Shorter block lengths were less effective primarily because such short system call traces were not particularly informative, as illustrated in Section 5.4.5. The primary advantage of the sequential detection methods is that they terminate as soon as enough evidence is collected that a process is malicious. Page's CUSUM offers the additional advantage that it is a continuous inspection scheme. That is, it monitors a process throughout its execution, and can detect the execution of malicious code regardless of when it occurs.

5.7 Conclusions

This section presented four detection algorithms and studied their empirical detection accuracy under fixed FPR constraints. The empirical results were used to determine the set of algorithms and parameters that provided the highest detection accuracy, guiding the design of a malicious process detection system. Logistic regression (LR) was chosen because it provided the highest detection accuracy and provided probability estimates as outputs. The selected feature extraction techniques included information retrieval, feature selection, and feature scaling techniques. The selected techniques used

- ordered 3-grams and trace lengths $l \geq 1000$ for information retrieval;
- a subset of 4,000 3-grams chosen using RFE for feature selection; and
- logarithmic frequency, IDF transformation, and L_2 normalization for feature scaling.

The selected detector achieved a TPR exceeding 0.95 at a FPR of 10^{-5}. Experimental results demonstrated the detector to be robust to drift and highly effective against specific malware families. A continuous inspection scheme using Page's CUSUM test enabled the rapid detection of malicious behaviors occurring at any time during the execution of a process.

Chapter 6: Malicious Process Classification

This chapter addresses the third major challenge of this thesis, that of classifying new malicious processes according to their behavioral similarity to known malware. Classification refers to the process of identifying the malware category or malware family to which a malicious process belongs. The malware classifier studied in this section uses the same system call trace data used for detection and performs classification as soon as a malicious process is detected. The outputs of the classifier are intended to guide mitigation and post-mortem analysis. Furthermore, the classifier outputs can be combined with other classification techniques to aid in detailed malware analyses [26]. The two primary challenges in malware classification are

- determining a set of techniques to quickly and accurately classify malicious processes using the same system call traces used for detection, and
- identifying a set of ground truth labels to use for training and testing.

To address the first challenge, Section 6.1 presents five classification algorithms selected for their previously demonstrated effectiveness in both document and malware classification. Section 6.2 describes the techniques used to evaluate the classifiers, and Section 6.3 presents the experimental results. The second challenge arises from the inconsistency and incompleteness of existing malware labeling systems. Since there is no universally accepted malware labeling scheme, Section 6.3.1 explores the use of multiple ground truth labeling schemes derived from antivirus (AV) signatures.

6.1 Malware classification algorithms

The classification techniques considered in this chapter are used in a supervised learning context, wherein the classification models are learned from labeled training data. This chapter considers five classification algorithms, the first two of which are multi-class implementations of detection algorithms presented in Chapter 5. The classification algorithms are

- a multi-class implementation of the logistic regression (LR) algorithm (Section 6.1.1);
- naïve Bayes, a multi-class implementation of the log-likelihood ratio test (LLRT) (Section 6.1.2);
- random forests of decision trees (Section 6.1.3);
- a nearest neighbors classifier (Section 6.1.4); and
- a nearest centroid classifier (Section 6.1.5).

6.1.1 Multi-class logistic regression (LR)

The LR training and testing algorithms described in Section 5.1.4 are designed for a binary decision task. Classification is a multi-class problem, wherein the goal is to identify to which of many families or categories a malware processes belongs. This study uses a multi-class implementation of LR known as the one-versus-all (OVA) approach [124]. The OVA approach treats the K-class classification problem as a collection of binary detection problems. The classifier uses K detectors, one for each of the malware classes. Each detector is trained to differentiate between the malware belonging to a specific malware class $\mathcal{C}_k$ and malware belonging to all other classes $\mathcal{C}_{\bar{k}}$.

The classifier selects the class with the highest decision statistic. Given $p_{\mathcal{C}_k}(\hat{\mathbf{x}})$, the probability that a process with feature vector $\hat{\mathbf{x}}$ comes from a malware sample of class $\mathcal{C}_k$, and $p_{\mathcal{C}_{\bar{k}}}(\hat{\mathbf{x}})$, the probability that it does not come from class $\mathcal{C}_k$, the OVA classifier selects the most likely class $\hat{\mathcal{C}}_k$ as

$$\hat{\mathcal{C}}_k = \underset{\mathcal{C}_k}{\arg\max} \;\; \log\left(\frac{p_{\mathcal{C}_k}(\hat{\mathbf{x}})}{p_{\mathcal{C}_{\bar{k}}}(\hat{\mathbf{x}})}\right). \tag{6.1}$$

The LR classifier is considered here because it offered the best performance of the four detection algorithms considered in the previous chapter, because of its computational simplicity during detection, and because of its efficient training process.

6.1.2 Naïve Bayes

Like the LR detector, the LLRT described in Section 5.1.2 is also a binary detection algorithm. The multi-class implementation of the LLRT is commonly referred to as the naïve Bayes algorithm, as it bases its detection thresholds on error costs and prior probabilities [116]. The classifier is based on

Bayes' rule, which states that the posterior probability is proportional to the product of the prior probability and likelihood. That is, the posterior probability $P(\mathcal{C}_k|\hat{\mathbf{x}})$ that a process comes from class $\mathcal{C}_k$ given its feature vector $\hat{\mathbf{x}}$ is proportional to the prior probability $P(\mathcal{C}_k)$ that a malware sample is from class $\mathcal{C}_k$ and the class-conditional likelihood of $\hat{\mathbf{x}}$,

$$P(\mathcal{C}_k|\hat{\mathbf{x}}) \propto P(\mathcal{C}_k) \times P_{\hat{X}|\mathcal{C}_k}(\hat{\mathbf{x}})\,. \tag{6.2}$$

The naïve Bayes classifier computes the most likely class label $\hat{\mathcal{C}}_k$ of a feature vector $\hat{\mathbf{x}}$ as the class with the highest posterior probability

$$\hat{\mathcal{C}}_k = \underset{\mathcal{C}_k}{\arg\max}\, P(\mathcal{C}_k) \times P_{\hat{X}|\mathcal{C}_k}(\hat{\mathbf{x}})\,. \tag{6.3}$$

For this study, the prior probabilities $P(\mathcal{C}_k)$ are estimated from the training data.

Two different implementations of the naïve Bayes classifier are considered in this work: a multinomial and a Gaussian implementation. The multinomial approach models the probabilities $P_{\hat{X}|\mathcal{C}_k}(\hat{\mathbf{x}})$ using a multinomial distribution and is used with the term frequency – inverse document frequency (TF-IDF) transformed feature data. The Gaussian approach models the posterior probabilities $P_{\hat{X}|\mathcal{C}_k}(\hat{\mathbf{x}})$ using a Gaussian distribution and is used with the singular value decomposition (SVD) and linear discriminant analysis (LDA) transformed feature data. Although it performed poorly for malware detection, the naïve Bayes algorithm is considered for classification because of its computational simplicity during training and testing and because related work has demonstrated its usefulness for malware classification.

6.1.3 Random forests

The random forests algorithm uses a collection of binary decision trees for classification. Given a the system call trace of a malware sample with a feature vector $\hat{\mathbf{x}}$, a binary decision tree classifies a process through a sequence of simple threshold tests. The random forest classifier is an ensemble learner that trains a collection of decision trees to use for classification. The random forests algorithm works by selecting the majority class $\hat{\mathcal{C}}_k$ output by the collection.

A decision tree comprises of a set of nodes and edges. The interior nodes of the tree correspond to simple threshold tests, each of which is performed against a single scalar feature $\hat{x}_i \in \hat{\mathbf{x}}$. During detection, the tree is traversed by evaluating the threshold test at each node, moving to the node's left child if $\hat{x}_i \leq \lambda$, or right child if $\hat{x}_i > \lambda$. The tree traversal continues until a leaf node is reached. Each leaf node of the tree represents a predicted malware class $\hat{\mathcal{C}}_k$, which is the output of the classifier.

The classification and regression trees (CART) decision tree algorithm was used in this study to train the decision tree classifier [125]. The CART algorithm builds decision trees recursively, starting at the root node of the tree. It begins by considering all of the malware samples and their corresponding labels and feature vectors $\hat{\mathbf{x}}$. It selects a feature $x_i \in \hat{\mathbf{x}}$ and a threshold λ that divide the data into two sets, those samples that exceed the threshold and those that do not. The CART algorithm seeks the feature and threshold that cause similar malware samples to be grouped together after the split. To achieve this goal, the feature and threshold are selected such that they minimize the weighted average of the Gini impurity of the resulting sets. The Gini impurity considers a set of feature vectors $\mathcal{X}$ each belonging to one of K classes, denoted $\mathcal{C}_k$. Given probabilities $p(\mathcal{C}_k)$ that a malware sample belong to class $\mathcal{C}_k$ estimated from the training data, the Gini impurity G of a set $\mathcal{X}$ is

$$G(\mathcal{X}) = \sum_{k=1}^{K} p(\mathcal{C}_k)\left(1 - p(\mathcal{C}_k)\right) . \tag{6.4}$$

The minimum Gini impurity (0) is achieved when a set contains only malware of a single class.

The feature and threshold are chosen for the root node divide the malware into two disjoint sets at the two child nodes. The process is repeated for each child node, and continues until a node contains only malware of the same class. Such nodes becomes leaf nodes of the decision tree. The algorithm terminates based on tunable parameters that determine the complexity of the tree, including the minimum number of training instances assigned to each leaf node and the minimum number of training instances required to split a node.

Decision tree models were selected for the white box decision models they provide. The random forest classifier [126], was selected due to the tendency of decision trees to overfit the training data. To ensure diversity in the decision trees, the decision tree training algorithm is modified to consider a only randomly selected subset of features at each node.

6.1.4 Nearest neighbors

The k-nearest neighbors classifier uses the entire training set $\mathcal{X}$ as its model [113]. For detection, it computes the Euclidean distance between a vector $\hat{\mathbf{x}}$ and every vector in the training set $\mathcal{X}$. The vectors in the training set nearest to $\hat{\mathbf{x}}$ are considered its nearest neighbors. The output of the classifier is the class with the highest representation in the set of nearest neighbors.

The nearest neighbor classifier is advantageous because of its simplicity and because it can realize complicated decision surfaces. However, its usefulness is limited to low-dimensional datasets due to the difficulty of comparing the distances among points in high dimensional space. Feature reduction techniques are typically used with the nearest neighbors classifier to project high-dimensional data onto a lower-dimensional space. In this study, SVD and LDA are used for feature reduction. Since it stores the entire training set as its model, the memory and computational complexity of the nearest neighbors classifier are too high to be considered for widespread deployment. Rather, the nearest neighbor classifier is considered here primarily for comparison.

6.1.5 Nearest centroid

The nearest centroid classifier models each malware class as its centroid, *i.e.*, the average of its feature vectors [127]. Classification is performed by computing the Euclidean distance from a feature vector $\hat{\mathbf{x}}$ to each centroid. The output of the classifier is the class label of the centroid nearest to the vector.

The nearest centroid classifier is advantageous because of its lower model and computational complexity compared to the nearest neighbor classifier. Whereas the model storage and computational complexity of the nearest neighbor classifier is $O(N)$, where N is the number of training samples, the storage and computational complexity of the nearest centroid classifier is $O(K)$, where K is the number of classes, and $K \ll N$. However, the nearest centroid classifier assumes convexity of its classes and equal variance along all dimensions. Therefore, it performs poorly when the data points within the classes do not form non-overlapping convex sets or when the feature variances differ significantly. Like the nearest neighbor classifier, the nearest centroid classifier is typically used with feature reduction techniques, such as the SVD and LDA transformations considered in this study.

6.2 Malware classifier evaluation

The techniques described in this section were used to evaluate the effectiveness of the selected malware classification algorithms. One set of techniques is used to evaluate classification accuracy on a per-class basis, while the other is used to asses the overall accuracy of a classifier. For per-class accuracy, this study considers precision, recall, and F_1 scores. For overall accuracy, it considers average F_1 scores and κ statistics. These per-class techniques consider the following four quantities for each class $\mathcal{C}_k$.

- $FP_{\mathcal{C}_k}$ is the number of false positives, *i.e.,* the number of processes incorrectly classified as belonging to class $\mathcal{C}_k$.
- $TP_{\mathcal{C}_k}$ is the number of true positives, *i.e.,* the number of processes correctly classified as belonging to class $\mathcal{C}_k$.
- $FN_{\mathcal{C}_k}$ is the number of false negatives, *i.e.,* the number of processes in $\mathcal{C}_k$ incorrectly classified as belonging to a different class.
- $TN_{\mathcal{C}_k}$ is the number of true negatives, *i.e.,* the number of processes not in $\mathcal{C}_k$ correctly classified as not belonging to class $\mathcal{C}_k$.

6.2.1 Precision

The precision of a class $\mathcal{C}_k$ is the fraction of processes classified as $\mathcal{C}_k$ that belong to $\mathcal{C}_k$,

$$\text{Precision}_{\mathcal{C}_k} = \frac{TP_{\mathcal{C}_k}}{TP_{\mathcal{C}_k} + FP_{\mathcal{C}_k}}\,. \tag{6.5}$$

The precision provides a measurement of the relevance of the positive classifications. A precision of 1 indicates that the classifier is always correct when it classifies a process as belonging to class $\mathcal{C}_k$, whereas a precision of 0 indicates it is never correct when it does so.

6.2.2 Recall

The recall of a class $\mathcal{C}_k$ is the fraction of the processes belonging to $\mathcal{C}_k$ in the ground truth that are correctly classified as $\mathcal{C}_k$,

$$\text{Recall}_{\mathcal{C}_k} = \frac{TP_{\mathcal{C}_k}}{TP_{\mathcal{C}_k} + FN_{\mathcal{C}_k}}\,. \tag{6.6}$$

The recall provides a measurement of the sensitivity of the classifier. A recall of 1 indicates that every instance of class $\mathcal{C}_k$ was correctly classified, whereas a recall of 0 indicates that every instance of $\mathcal{C}_k$ was incorrectly classified.

6.2.3 F_1 score

The F_1 score of a class $\mathcal{C}_k$ is the harmonic mean of the precision and recall,

$$F_{1,\mathcal{C}_k} = 2\,\frac{Precision_{\mathcal{C}_k} \times Recall_{\mathcal{C}_k}}{Precision_{\mathcal{C}_k} + Recall_{\mathcal{C}_k}}. \tag{6.7}$$

An F_1 score of 0 indicates 0 recall or 0 precision, whereas an F_1 score of 1 indicates perfect recall and precision. In this study, F_1 score is used to evaluate classification accuracy both per-class and as an aggregate measure over all the classes. To evaluate the overall F_1 score, the individual class scores are averaged over the classes. Three different averaging techniques are considered.

Micro-averaged F_1 score: Each process is given equal weight, and the average is computed over the individual processes. This provides an indicator of how effective the classifier is on large classes, since classes with higher representation receive the most weight.

Macro-averaged F_1 score: Each class is given equal weight, and the average is the average of the F_1 scores from all the classes. This provides an indicator of how effective the classifier is on small classes, since all classes are treated equally.

Weighted F_1 score: Each class is weighted inversely by its support in the ground truth. This approach is intended to give a balanced measure of the F_1 score when classes are not balanced, as is the case in this work.

All three averaging techniques are considered because the classes of the malware samples used in this study are not balanced. By including all three values, the performance of the classifiers against small classes, large classes, and on average can be studied.

6.2.4 Cohen's κ statistic

The F_1 score does not take into account the effects of chance on classification performance. To address this concern, this study also uses Cohen's κ statistic for classifier evaluation [128]. The κ statistic is intended for the comparison of two different sets of class assignments and provides a measure of the agreement between the sets. Here, the κ statistic is used to measure the agreement between classification results and the ground truth. The κ statistic compares the accuracy of a classifier to the accuracy of a random classifier operating on the same dataset. The κ statistic compares the observed accuracy of a classifier to the expected accuracy of a random classifier. The observed accuracy of a classifier is the fraction of processes it correctly classifies,

$$ACC_{\text{observed}} = \frac{1}{\mathbf{card}(\mathcal{X})} \sum_{\mathcal{C}_k} TP_{\mathcal{C}_k} \,. \tag{6.8}$$

The expected accuracy of a classifier is computed by estimating the probability of occurrence of each class separately for the ground truth and classification results. The expected accuracy assumes the class assignments made by both the classifier and the ground truth are randomly determined according to the estimated probabilities of each class. Given $p_{\text{cr}}(\mathcal{C}_k)$ as the probability of appearance of each class $\mathcal{C}_k$ in the classification results, and $p_{\text{gt}}(\mathcal{C}_k)$ as the probability of appearance of each class $\mathcal{C}_k$ in the ground truth, the expected accuracy is

$$ACC_{\text{expected}} = \sum_{\mathcal{C}_k} p_{\text{cr}}(\mathcal{C}_k) \times p_{\text{gt}}(\mathcal{C}_k) \,. \tag{6.9}$$

The κ statistic is computed from the observed and expected accuracy as

$$\kappa = \frac{ACC_{\text{observed}} - ACC_{\text{expected}}}{1 - ACC_{\text{expected}}} \,. \tag{6.10}$$

The κ values range from -1 to 1, with 1 indicating perfect agreement, -1 indicating perfect disagreement, and 0 indicating the results are not discernible from random chance. The κ statistic is useful in this study because the class imbalance of the malware samples (especially for categorical labels) contributes to a high expected accuracy of a random classifier.

6.3 Experimental classification results

This section presents experimental results achieved using the classifiers described in Section 6.1 and evaluation techniques described in Section 6.2. The results were obtained through cross-validation performed on the set of processes collected from the execution of more than 55,000 distinct malware samples. Among the experimental results described in this section are

- a comparison of the classification accuracy afforded by multiple ground truth labeling schemes (Section 6.3.1);
- an evaluation and comparison of the five classification algorithms (Section 6.3.2);
- analysis of per-category and per-family classification accuracy (Sections 6.3.3 and 6.3.4); and
- a feature selection study, comparing the sets of features chosen for detection and classification (Section 6.3.5).

The outcome of this section is the design of a malware classification system, with techniques and parameters chosen experimentally to maximize its accuracy against the malware samples considered for this study.

6.3.1 Ground truth comparison

The lack of a universal malware naming scheme, combined with the incompleteness and inconsistency of existing naming schemes, complicates the problem of automatic classification. In particular, it raises the question: What are the relevant malware classes against which a classifier should be evaluated? This section presents a study of the accuracy of the LR classifier against 27 different ground truth labeling schemes derived from 16 different AV vendors' labels. The ground truth labeling schemes are listed in Table 6.1, identified by the AV labels from which they are derived and

Table 6.1: Classification ground truth performance comparison for the LR classifier

Ground truth	classes	processes	κ	F_1 (micro)	F_1 (macro)	F_1 (weighted)
AntiVir-category	16	62,398	0.74	0.84	0.51	0.83
AntiVir-family	167	59,478	0.59	0.62	0.40	0.61
Avast-category	12	78,147	0.51	0.64	0.40	0.67
Avast-family	192	75,146	0.50	0.52	0.38	0.53
DrWeb-category	9	74,256	0.61	0.77	0.41	0.79
DrWeb-family	226	70,857	0.62	0.62	0.44	0.64
Emsisoft-category	32	77,284	0.49	0.57	0.30	0.59
Emsisoft-family	258	68,434	0.47	0.48	0.32	0.50
ESETNOD32-family	251	75,079	0.68	0.69	0.48	0.70
FSecure-category	46	75,726	0.58	0.66	0.47	0.66
FSecure-family	172	72,331	0.55	0.57	0.39	0.57
GData-category	26	81,038	0.52	0.60	0.32	0.62
GData-family	218	72,807	0.48	0.50	0.38	0.51
Ikarus-category	41	75,429	0.54	0.58	0.40	0.60
Ikarus-family	419	67,937	0.55	0.56	0.38	0.55
K7AntiVirus-category	14	73,229	0.56	0.66	0.51	0.67
Kaspersky-family	259	69,325	0.53	0.54	0.40	0.55
Kaspersky-category	30	72,933	0.56	0.61	0.39	0.63
McAfee-category	111	79,477	0.55	0.57	0.48	0.58
Microsoft-family	263	61,883	0.72	0.73	0.53	0.73
Microsoft-category	19	66,930	0.71	0.75	0.50	0.76
Panda-category	8	71,352	0.60	0.76	0.51	0.76
Panda-family	98	68,838	0.56	0.60	0.47	0.60
Symantec-category	22	68,019	0.52	0.62	0.41	0.62
TrendMicro-family	207	57,950	0.50	0.52	0.40	0.52
Vipre-category	28	77,032	0.58	0.71	0.39	0.74
Vipre-family	185	74,713	0.52	0.58	0.41	0.61

whether they are category or family labels. For example, the AntiVir-category labels are categorical labels derived from the AntiVir AV labels. For each of the ground truth labeling schemes, the table lists the number of classes and total number of processes included in the labeling scheme. For each set of labeling schemes, only malware samples positively identified as malicious by the corresponding AV vendor were considered. Furthermore, only classes containing at least ten malware samples were considered to ensure meaningful cross-validation results.

Table 6.1 presents the performance of the LR classifier in terms of both its κ and F_1 scores. The results provide two useful insights into the performance of the classifier. First, the macro-averaged F_1 score is consistently low, indicating the classifier performs poorly against small classes regardless of the ground truth. Second, the κ scores, micro-averaged F_1 scores, and weighted F_1 scores varied significantly based on the chosen ground truth. This observation indicates that system call 3-gram features characterize certain ground truth labeling schemes better than others. The highest performing categorical ground truth labels were those derived from the AntiVir and Microsoft labels, while the highest performing family ground truth labels were those derived from the ESET and Microsoft labels. The high classification performance of the AntiVir labels was attributed to a single class accounting for the vast majority of the samples. Therefore, only the ESET and Microsoft labels are considered for the remainder of this study.

6.3.2 Classifier comparison

This section compares the accuracy of the five classifiers described in Section 6.1 using three different feature extraction strategies.

- The first strategy (TF-IDF) considers the best performing feature set from the detection results, namely logarithmic frequencies, inverse document frequency (IDF) transformation, and L_2 scaling.
- The second strategy (TF-IDF, SVD) combines the first with SVD, wherein the TF-IDF transformed features are projected onto their first 500 singular values.
- The third strategy (TF-IDF, SVD, LDA) combines the second with LDA, wherein the SVD transformed features are projected onto $K - 1$ features selected using LDA.

While the LR classifier is evaluated using all three feature extraction strategies, the other classifiers are evaluated using only a subset of the techniques. The nearest centroid, nearest neighbor, and decision tree algorithms do not consider the TF-IDF features because of the high dimensionality of the data and the computational complexity of the algorithms. The multinomial and Gaussian naïve Bayes are restricted only to the feature sets that can be described by the multinomial and Gaussian distribution models, respectively. Therefore, the former uses only the TF-IDF data and the latter uses only the SVD and LDA transformed feature data.

Table 6.2 summarizes the performance of each classifier and feature extraction strategy combination in terms of its κ statistic and averaged F_1 scores. Among the worst performers were the the naïve Bayes and nearest centroid classifiers. The naïve Bayes classifier performed poorly because the

Table 6.2: Classifier and feature extraction comparison

detector	feature extraction	κ	F_1 (micro)	F_1 (macro)	F_1 (weighted)
LR	TF-IDF	0.73	0.73	0.55	0.73
LR	TF-IDF, SVD	0.60	0.61	0.35	0.62
LR	TF-IDF, SVD, LDA	0.66	0.66	0.44	0.66
nearest centroid	TF-IDF, SVD	0.29	0.29	0.17	0.34
nearest centroid	TF-IDF, SVD, LDA	0.55	0.55	0.32	0.58
nearest neighbor	TF-IDF, SVD	0.72	0.72	0.47	0.71
nearest neighbor	TF-IDF, SVD, LDA	0.73	0.73	0.49	0.72
random forests	TF-IDF, SVD	0.73	0.73	0.52	0.72
random forests	TF-IDF, SVD, LDA	0.72	0.73	0.50	0.71
multinomial naïve Bayes	TF-IDF	0.42	0.43	0.06	0.34
Gaussian naïve Bayes	TF-IDF, SVD	0.47	0.47	0.35	0.52
Gaussian naïve Bayes	TF-IDF, SVD, LDA	0.47	0.47	0.35	0.52

assumed probability models were inaccurate and tended to overfit the training data. The nearest centroid classifier performed poorly because the feature data did not form non-overlapping convex sets. The nearest centroid detector's performance improves significantly when LDA is used, because the resulting sets have higher separation. However, it still significantly underperforms the highest performing classifiers.

The highest performers were the nearest neighbor, random forest, and LR classifiers. The first two performed equally well whether SVD or LDA was used. The high accuracy of the nearest neighbor classifier underscores the weakness of its counterpart, the nearest centroid classifier. Although the two use the similar distance metric-based algorithms, the simplistic model and convexity assumption of the nearest centroid classifier causes its poor performance. The relatively poor performance of the LR classifier with feature reduction indicates that the feature data are not linearly separable in the reduced feature space. The nearest neighbor and random forests classifiers achieved accurate results due to the nonlinear decision surfaces they are able to realize.

For a production deployment, the LR classifier is preferred to the random forests and nearest neighbor algorithms. The LR classifier provides lower training complexity than the random forest classifier and provides an efficient mechanism for re-training the models as new malware samples are discovered. The LR classifier also provides a lower testing complexity than the nearest neighbor

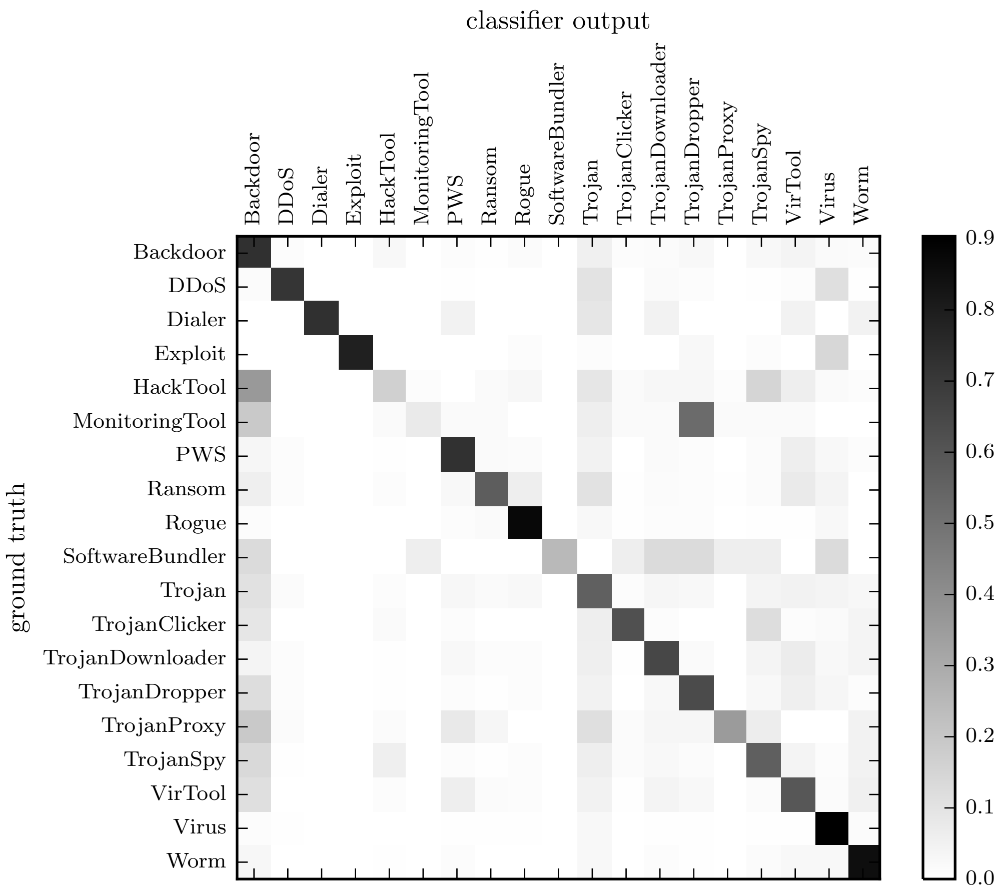

Figure 6.1: Classifier confusion matrix for Microsoft category ground truth labels, showing the fraction of samples in each class indicated by the row labels classified as the column labels

classifier, making it more desirable for deployment on production hosts.

6.3.3 Category-level classification results

The preceding sections provided a high-level overview of classification performance in terms of κ statistics and averaged F_1 scores. This section provides analysis of per-category accuracy of the LR classifier using the Microsoft category labels. The experimental results are presented in terms of the classifier's confusion matrix. The confusion matrix is a $K \times K$ matrix, wherein the rows represent the ground truth labels and the columns represent the classifier outputs. Traditionally, the entries of a confusion matrix are the number of instances of the row class assigned to the column class by the classifier. The diagonal entries indicate correctly classified instances, while the non-diagonal entries indicate incorrectly classified instances. Figure 6.1 shows the confusion matrix for the LR classifier using the Microsoft family labels. Since the classes are not balanced, the entries of the matrix are the percentage of the row class assigned to the column class by the classifier. For example, more than 90% of the worms were correctly classified as worms, whereas less than 20% of the monitoring tools were correctly classified as monitoring tools.

The confusion matrix shows two dark vertical lines in the backdoor and Trojan columns, indicating that many malware samples from multiple categories were misclassified as backdoors and Trojans. The backdoor misclassifications were caused by malware samples in other classes providing backdoors and backdoor-like functionality. For example, many malware samples used in this study notified remote servers when a host is infected, sent confidential data to remote servers, or requested commands and configuration information from remote servers. Conversely, the Trojan misclassifications were caused by the abundance of Trojans in the training set and the wide variety of functionality they exhibited. The Trojans used in this study provided functions that overlapped with the majority of the other malware classes. Many Trojans included backdoor, distributed denial of service (DDoS), password stealing, and other malicious functions.

The confusion matrix in Figure 6.1 also shows a dark horizontal line on the software bundler row, which indicates that many software bundler samples were misclassified. The software bundler samples were most often misclassified as various types of Trojans. This is also likely due to the nature of a the software bundlers, which are legitimate software that also install malware. Whereas a Trojan is disguised as legitimate software, a software bundler actually contains legitimate software as the delivery mechanism for the malware. The misclassifications can be attributed to this similarity in function and to the fact that Trojans outnumbered software bundlers in the malware set.

There were two other categories for which the classifier performed particularly poorly, hack tools and monitoring tools. Hack tools include key generators and password crackers. Such tools are often distributed with other bundled malware, which likely led to the misclassifications. Monitoring tools are commercial programs that monitor computer usage, and can include keyloggers and password stealers. Monitoring tools are similar to Trojans and Trojan Droppers, because they typically are bundled with and installed alongside other software. They are similar to spyware because they transmit collected data over the network. Overall, the results presented in this section indicate that overlapping malware functionality and broadly-defined malware families are the primary cause of category-level misclassifications.

6.3.4 Family-level classification results

Whereas the previous section characterized the performance of the LR classifier using the Microsoft category labels, this section characterizes its performance using the Microsoft family labels. In this section, the experimental results are presented in terms of the precision, recall, and F_1 score for each of the malware categories in the ground truth. The results presented in Table 6.3 show the precision, recall, F_1 score, and number of instances of the 25 malware families with the highest F_1 scores, and the 25 malware families with the lowest F_1 scores. Among the best-performing families are those with thousands of instances, whereas the poorest performers have at most 69 instances. These results are consistent with those presented in Section 6.3.2, which indicated that the classifier performs well for classes with many instances and poorly for classes with few instances.

Generally, the best performing families were those from narrowly defined malware categories, such as Viruses, worms, exploits, password stealers (PWSs), and backdoors. Conversely, the worst performing families were those from more broadly defined categories, particularly Trojans. Furthermore, some of the poorly performing families are also broadly defined. For example, Trojan.Keylogger, TrojanDropper.VB, and Trojan.Bumat!rts are generic families for keyloggers, Trojan droppers, and Trojans. In contrast, the highest performing families are typically very narrowly defined. For example, the Worm.Klez definition specifies the registry entries and files modified by the malware, its interaction with other malware and AV software, how it searches for email addresses, the contents of the emails it sends, how it propagates, and the security vulnerabilities it exploits.

One way to account for varying class sizes when training a classifier is to weight the classes inversely based on their representation in the training set. Rare classes are oversampled to even the class distributions and ensure that the classifier is not biased toward classes with higher representation. For comparison, two versions of the LR classifier were trained, one weighting the classes

Table 6.3: Per-family classifier precision, recall, and F_1 scores for the malware families with the highest and lowest F_1 scores

Family	Precision	Recall	F_1 score	Instances
Virus.Nabucur	1	1	1	1652
Worm.Klez	1	1	1	218
Trojan.Recal	1	1	1	331
Worm.Ganelp	1	1	1	24
Worm.Fesber	1	1	1	702
Worm.Dumpy	1	1	1	18
Worm.Mydoom	0.99	1	0.99	1985
Backdoor.Wabot	0.98	0.99	0.99	324
TrojanDownloader.Seimon	0.98	1	0.99	47
TrojanDropper.Loring	0.97	0.99	0.98	706
Virus.Madang	0.98	0.97	0.98	703
Worm.VB	0.97	0.99	0.98	2423
TrojanDownloader.Ogimant	0.99	0.98	0.98	1202
Worm.Krol	0.95	0.98	0.97	43
Exploit.RpcDcom	0.93	1	0.96	25
Trojan.Startpage	0.96	0.96	0.96	921
PWS.Uosproy	0.95	0.98	0.96	309
Worm.Benjamin	0.96	0.96	0.96	24
Trojan.Phishbank	0.92	0.98	0.95	198
Backdoor.Small	0.93	0.94	0.94	327
Virus.Ipamor	1	0.88	0.94	17
Worm.Lightmoon	0.89	1	0.94	65
TrojanDownloader.Kanav	0.89	0.98	0.94	59
Worm.SillyFDC	0.89	0.99	0.94	89
Virus.Ramnit	0.99	0.87	0.93	2983
...				
Ransom.Dircrypt	0.08	0.11	0.09	28
Backdoor.Bergat	0.27	0.05	0.09	55
Ransom.Urausy	0.05	0.14	0.08	28
Trojan.Rimecud	0.05	0.29	0.08	17
TrojanDropper.Cutwail	0.07	0.1	0.08	10
TrojanDownloader.Renos	0.07	0.07	0.07	14
Trojan.Agent	0.05	0.14	0.07	28
Trojan.Meredrop	0.07	0.07	0.07	41
Trojan.Sisproc	0.06	0.06	0.06	32
PWS.Ldpinch	0.06	0.05	0.06	20
TrojanDropper.VB	0.04	0.07	0.05	68
TrojanSpy.VB	0.03	0.15	0.05	20
Trojan.Bumat!rts	0.03	0.04	0.04	69
Trojan.Sisron!gmb	0.04	0.04	0.04	46
Trojan.Nedsym	0.03	0.06	0.04	16
Trojan.Orsam!rts	0.02	0.05	0.03	44
TrojanDownloader.Umbald	0.06	0.02	0.03	41
Trojan.Anomaly	0	0	0	42
Trojan.Napolar	0	0	0	14
Trojan.Yakad	0	0	0	24
Trojan.Comame	0	0	0	23
Trojan.Rimod	0	0	0	16
TrojanSpy.Keylogger	0	0	0	62
Trojan.Anaki	0	0	0	21
TrojanDownloader.Delf	0	0	0	13

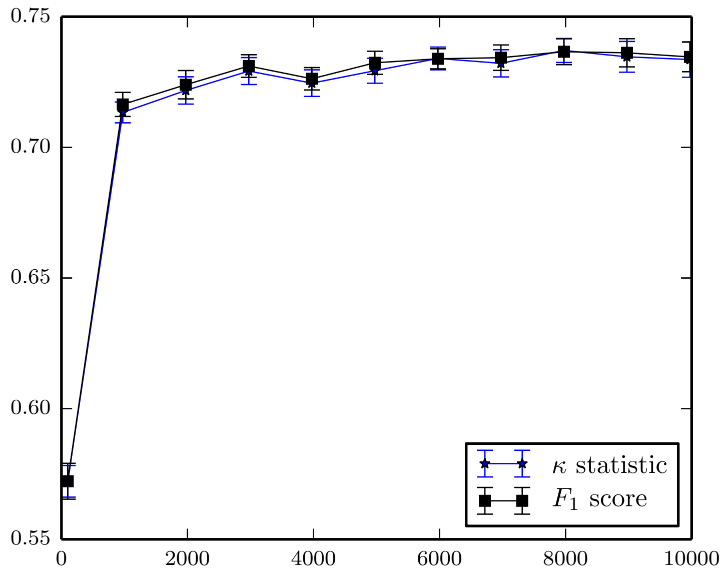

Figure 6.2: Effects of feature selection on classification accuracy, showing the κ statistic and weighted F_1 scores of the classifier versus the number of features selected using recursive feature elimination (RFE)

inversely based on their representation and the other using the training-data without modification. The two versions of the classifier provided nearly identical results, including the same κ and weighted F_1 scores. The similarity of the two classifiers indicate that the size of the samples was not biasing the classifier. Rather, the low precision and recall of the worst-performing classes in Table 6.3 were likely caused by broadly defined malware families, mislabeled ground truth, and a lack of enough samples to accurately characterize the families. Mislabeling may have occurred in the ground truth where malware samples installed additional samples from other families. However, due to limitations in the testing platform, the installed malware samples shared the ground truth labels of their parent malware, even when they belonged to a different family.

6.3.5 Feature selection

RFE was used with the LR classifier to identify the most informative 3-grams for the classification task. Figure 6.2 shows the cross validation results of RFE, showing the κ statistic and weighted F_1 scores for the classifier versus the number of chosen features. Similar to the detection results presented in the previous chapter, the classification accuracy increases rapidly approaching 1000 features and levels out as the number of features increases. The maximum classification κ and F_1 scores occur at 8000 features, much higher than the maximum detection accuracy, which occurred at 3000 features. This result indicates that more features are required for the classification than are required for detection.

A comparison of the 4000 highest ranked classification features and the 4000 features chosen for detection indicates 34% overlap. The overlap indicates that many of the same features used to distinguish malware from benign software are also useful for distinguishing among different classes of benign software. When only the 4000 features used for detection are used for classification, the LR classifier achieves a κ and weighted F_1 score of 0.70 and 0.71, respectively. This corresponds to a 4.1% decrease in classification accuracy when compared to the LR classifier when all the features are used.

6.4 Conclusions

This chapter explored the use of multiple classification algorithms, feature extraction strategies, and ground truth labeling schemes for malware classification. Through experimental evaluation, it demonstrated that decision tree, nearest neighbor, and LR classifiers outperformed naive Bayes and nearest centroid classifiers. Since the goal of this study was to identify those techniques best suited for production deployment, the LR classifier was selected for its comparatively low model and computational complexities. However, the LR classifier requires a larger feature set than the detector to achieve maximum accuracy. In practice, the set of features used by the detector and classifier would be expanded to afford both high detection and high classification accuracy. The results presented in this section also demonstrate that, among the studied malware labeling systems, the ESET and Microsoft naming systems afforded the highest classification accuracy. The classifier performed best against well-defined malware families from those labeling systems, and worst against broadly defined malware families and categories with few training instances.

Chapter 7: Conclusions

This study sought to determine the most effective feature extraction, detection, and classification techniques to use for the detection and classification of malicious processes based on their system call traces. The goal of this study was to identify techniques that were 'lightweight' enough and that exhibited a low enough false positive rate (FPR) to be useful in production environments.

7.1 System design

Through experimental evaluation, this study identified the set of techniques that provided the best detection performance at a fixed FPR of 10^{-5} and the highest classification accuracy. The best performing feature extraction techniques were

- using a minimum system call trace length of $l = 1{,}000$;
- using a bag-of-ordered-3-grams representation to represent system call traces;
- using a subset of 8,000 ordered 3-grams chosen using recursive feature elimination (RFE); and
- performing feature scaling using term frequency – inverse document frequency (TF-IDF) transformation with logarithmic term frequencies and L_2 normalization.

For detection, the logistic regression (LR) and support vector machine (SVM) algorithms offered the highest accuracy, outperforming a signature-based detector and log-likelihood ratio test (LLRT). The LR detector was chosen because its probability estimates enable its integration with the two sequential detection procedures considered in this study, Wald's sequential probability ratio test (SPRT) and Page's cumulative sum (CUSUM) test. Of the two sequential procedures, Page's CUSUM test is preferred because it enables the continuous inspection processes throughout their execution, and can detect latent malware that delay the execution of their malicious code.

For classification, the random forest, nearest neighbor, and LR algorithms provided the highest accuracy, outperforming naïve Bayes and a nearest centroid classifier. The LR algorithm was chosen

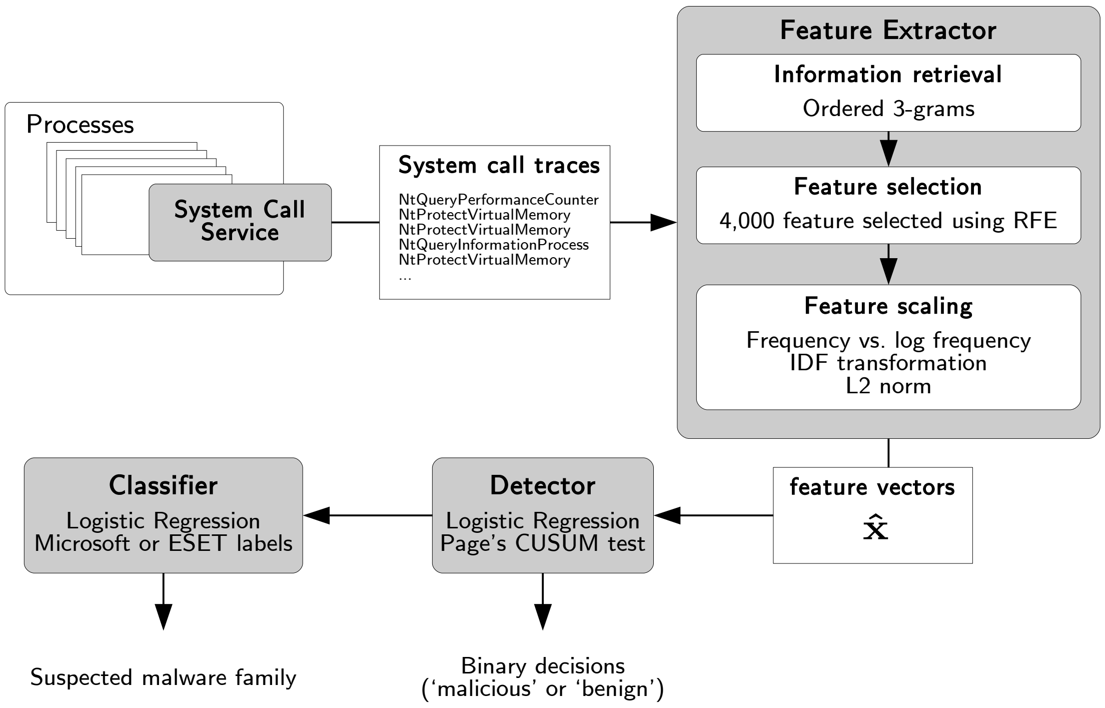

Figure 7.1: Block diagram of malware detection and classification system showing chosen feature extraction, detection, and classification techniques

over the nearest neighbor methods for its lower memory and computational complexity during testing and chosen over the random forests because of its lower memory and computational complexity during training.

A block diagram of the complete malware detection and classification system is provided in Figure 7.1. The figure shows the four major components of the system, the System Call Service (SCS), feature extractor, detector, and classifier, in gray. The SCS outputs system call traces which the feature extractor processes and represents as feature vectors $\hat{\mathbf{x}}$. The feature vectors are processed by the detector and classifier, which provide outputs indicating the suspected family of any detected malicious processes. The specific algorithms and parameters used by each component are indicated in the block diagram.

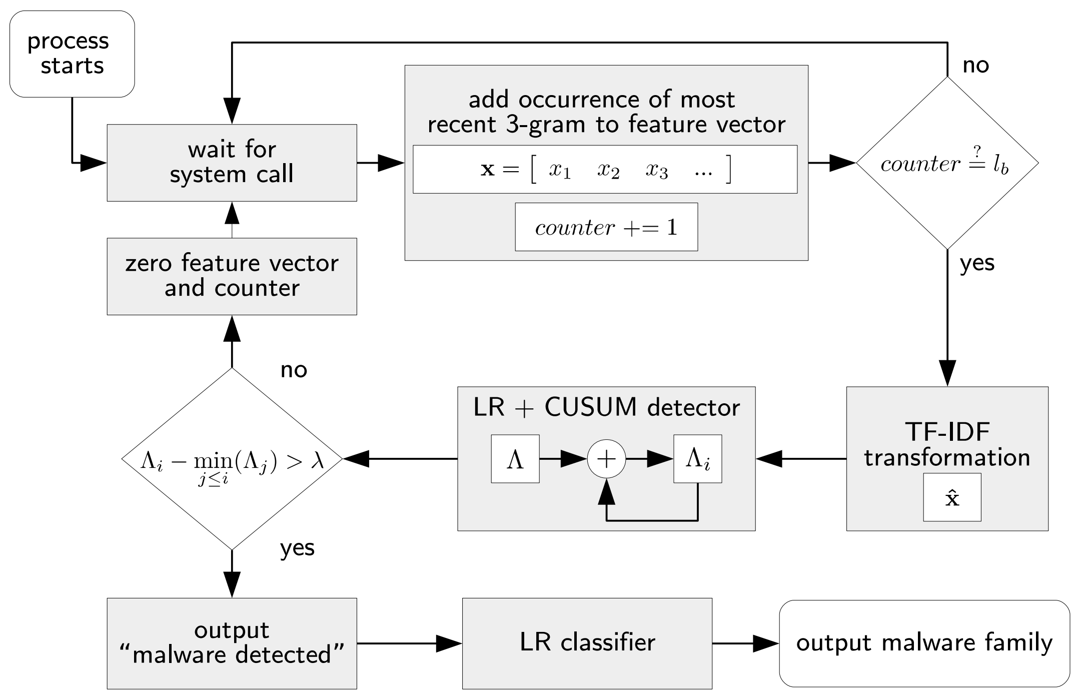
process starts
wait for system call
add occurrence of most recent 3-gram to feature vector
$\mathbf{x} = [\ x_1 \quad x_2 \quad x_3 \quad \dots\ ]$
$counter \mathrel{+}= 1$
$counter \overset{?}{=} l_b$
no
yes
zero feature vector and counter
TF-IDF transformation
$\hat{\mathbf{x}}$
LR + CUSUM detector
Λ
$+$
Λ_i
$\Lambda_i - \min_{j \le i}(\Lambda_j) > \lambda$
no
yes
output "malware detected"
LR classifier
output malware family

Figure 7.2: Flow chart showing specifics of detector and classifier operation

7.2 Summary of system operation

The operation of the detection and classification system is summarized by the flow chart in Figure 7.2. The flow chart shows the entire detection and classification process performed by the system as it monitors a process. The first row of the flow chart depicts the feature extraction loop. In this loop, the system waits for system call events to arrive from the process and counts the occurrences of the system call 3-grams. This process continues until the desired number of system calls l_b have been observed. Next, the feature extraction process is completed using TF-IDF transformation. The resulting feature vector $\hat{\mathbf{x}}$ is used for detection. From the output Λ of the LR detector, the sequential decision statistic Λ_i is computed as the cumulative sum of the LR outputs. If the decision statistic exceeds the detection threshold λ, the system performs classification using the saved feature vectors and the LR classifier. The classifiers outputs the malware family to which the malicious process most likely belongs. Otherwise, the detector re-enters the feature extraction loop and continues monitoring the process.

7.3 Discussions

Detection

This study sought to address two perceived shortcoming in the literature in behavioral malware detection. First, it sought to address conflicting claims regarding the effectiveness of proposed behavioral malware detection techniques. Particularly, it sought to address conflicting claims about the effectiveness of system call frequencies and sequences as feature sets for malware. Further, it sought to address conflicting claims regarding the usefulness of signature-based and statistical detection algorithms. In both cases, these claims were addressed by evaluating the effectiveness of such techniques against a common dataset. This study explored the feature space in which system call traces can be represented, providing empirical results to evaluate the discrimination afforded by each representation.

Second, this study sought to address the apparent lack of a study of behavioral malware detection performance at very low FPRs. Empirical evaluation of the detectors was performed using a set of 4 million processes and 55,000 malware samples. This study revealed that high detection accuracy was possible at a FPR as low as 10^{-5}, identifying the set of feature extraction and detection techniques that could be used to achieve such accuracy.

The method by which the system call traces of malware and benign processes used in this study were collected is also one of the main contributions of this thesis. Instead of executing malware in a specialized sandbox environment, malware were executed on live hosts configured to mimic production environments. Furthermore, the same techniques were used to collect benign process traces both on the same hosts as the malware and in production environments. This approach was used to prevent bias in the detection results arising from the use of specialized environments.

The majority of the malware samples in this study were detected in their early stages of execution. If the techniques described in this thesis were to be widely adopted, the most common form of evasion used by malware authors would likely be to delay the malicious tasks that lead to positive detections. To address this concern, this study presented a sequential malware detection procedure adapted from Page's CUSUM test. The proposed sequential detection procedure is a continuous inspection scheme that monitors each process for the duration of its execution. Thus, the test can detect malicious behaviors regardless of when they occur. This sequential malware detection procedure might also prove useful in detecting other types of faults and intrusions that do not necessarily occur at the beginning of a process's execution.

Classification

For malware classification, the contributions of this study were two-fold. First, it showed that classification could be achieved using the same set of system call 3-gram features used for malware detection. This enables classification to be performed immediately after a malware sample is detected, without needing to perform any additional data collection. Furthermore, the computationally simple algorithms used for classification enable rapid classification, even on low-end hardware. Second, it showed that the ground truth labels used for training and testing can have a significant effect on classification accuracy. Through the study of some 27 ground truth labeling schemes derived from 16 different antivirus (AV) vendors' labels, the labeling schemes providing the highest classification accuracy were identified.

Practical considerations

This study considered some of the practical considerations of the deployment of the described system, including the effects of drift on detection accuracy. In a commercial deployment of this system, it is expected that the models used for detection and classification would require periodic updates to be pushed to clients from a centralized server. Figure 7.3 depicts the model update process. Feature data, decisions, and malware samples collected at the client would be provided as feedback to the server. The server would use this information to either update the models directly, or perform additional analyses of the new malware samples. The updated models would be provided by the server to the client.

Whereas AV software signature databases are typically updated multiple times each day, the described system would require less frequent updates – on the order of weeks – to maintain its high accuracy. The cost of updating the models used for detection comes from the processes of

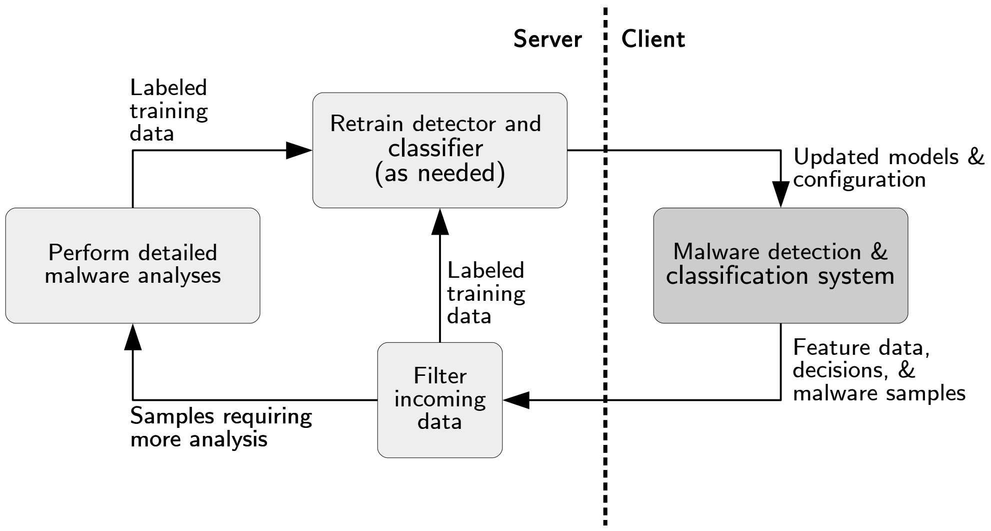
Server
Client
Labeled training data
Retrain detector and classifier (as needed)
Updated models & configuration
Perform detailed malware analyses
Labeled training data
Malware detection & classification system
Feature data, decisions, & malware samples
Filter incoming data
Samples requiring more analysis

Figure 7.3: Model update feedback loop, showing how model updates are generated and pushed to the clients using data collected from the clients

obtaining training data and learning the models. The former cost is low because the clients provide feature data, malware samples, and classification results that can be used to inform the model update process. This is advantageous because it does not require that new samples be executed in a laboratory environment. The provided feature data and classification results can also be used to guide additional analyses. Such analyses include the execution of the malware samples in a laboratory environment to determine whether the set of monitored features requires any updates. Such updates are expected to be rare, but may be necessary to accurately detect and classify new malware families. The updated configuration information would be pushed to the clients along with the updated models. The latter cost, that of learning the models, is low due to the chosen detection and classification training algorithm, stochastic gradient descent (SGD). The SGD algorithm is an efficient algorithm for training linear classifiers that performs well on large numbers of training samples and on high-dimensional data. Furthermore, SGD supports the successive refinement of models as additional training instances are added.

Finally, while this study focused on the detection and classification of malware processes, the techniques, test procedures, and SCS were developed with thread, application, and host-level detection in mind. Although not presented in this thesis, the described techniques have demonstrated promising results at the host-level, and preliminary work at the thread-level exhibited comparable detection performance. The advantage of performing detection at the thread level is the possible identification of specific malicious threads that execute in the context of an otherwise benign process. The advantage of performing detection at the application or host level is the possible identification of malicious behaviors being performed by a collection of threads or processes, such as those performed in a shadow attack.

Bibliography

[1] D. Plohmann and E. Gerhards-Padilla. Case study of the miner botnet. In *4th International Conference on Cyber Conflict*, CYCON, pages 1–16, 2012.

[2] S. Khattak, N.R. Ramay, K.R. Khan, A.A. Syed, and S.A. Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys Tutorials*, 16:898–924, 2014.

[3] Alexandre Gazet. Comparative analysis of various ransomware virii. *Journal in Computer Virology*, 6(1):77–90, 2010.

[4] Brett Stone-Gross, Ryan Abman, Richard A Kemmerer, Christopher Kruegel, Douglas G Steigerwald, and Giovanni Vigna. The underground economy of fake antivirus software. In *Economics of Information Security and Privacy III*, pages 55–78. Springer, 2013.

[5] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, 2005.

[6] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *International Symposium on Recent Advances in Intrusion Detection*, RAID, 2007.

[7] Carlos Castillo, Alex Hinchliffe, Chris Miller, Rajesh Nataraj KP, Francois Paget, Eric Peterson, Arun Pradeep, Craig Schmugar, Rick Simon, Dan Sommer, Bing Sun, and Adam Wosotowsky. Mcafee labs threats report. Technical report, Intel Security, 2015.

[8] Carey Nachenberg. Computer virus-antivirus coevolution. *ACM Communications*, 40(1):46–51, 1997.

[9] S. Neuhaus and T. Zimmermann. Security trend analysis with cve topic models. In *International Symposium on Software Reliability Engineering*, ISSRE, pages 111–120, IEEE, 2010.

[10] Sherly Abraham and InduShobha Chengalur-Smith. An overview of social engineering malware: Trends, tactics, and implications. *Technology in Society*, 32(3):183–196, 2010.

[11] A.K. Sood, R. Bansal, and R.J. Enbody. Cybercrime: Dissecting the state of underground enterprise. *IEEE Internet Computing*, 17(1):60–68, 2013.

[12] Steve Mansfield-Devine. A patchy response: the dangers of not keeping our systems secure. *Computer Fraud & Security*, 2015(1):15–20, 2015.

[13] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security Privacy*, 9(3):49–51, 2011.

[14] Beth Binde, Russ McRee, and Terrence J O'Connor. Assessing outbound traffic to uncover advanced persistent threat. Technical report, SANS Institute, 2011.

[15] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the analysis of the zeus botnet crimeware toolkit. In *Annual International Conference on Privacy Security and Trust*, PST, pages 31–38, 2010.

[16] Aditya K. Sood and Richard J. Enbody. Crimeware-as-a-service: A survey of commoditized crimeware in the underground market. *International Journal of Critical Infrastructure Protection*, 6(1):28–38, 2013.

[17] Brett Stone-Gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna. The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, LEET, 2011.

[18] Kregg Aytes and Terry Connolly. Computer security and risky computing practices: A rational choice perspective. *Journal of Organizational and End User Computing*, 16(3):22–40, 2004.

[19] Adam M Bossler and Thomas J Holt. On-line activities, guardianship, and malware infection: an examination of routine activities theory. *International Journal of Cyber Criminology*, 3(1): 400–420, 2009.

[20] Chris J. Mitchell, editor. *Trusted Computing*. The Institution of Engineering and Technology, 2005.

[21] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly, 2004.

[22] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[23] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Conference on system administration*, LISA, pages 229–238, USENIX Association, 1999.

[24] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28: 18–28, 2009.

[25] A.G. Tartakovsky, B.L. Rozovskii, R.B. Blazek, and Hongjoong Kim. A novel approach to detection of intrusions in computer networks via adaptive sequential and batch-sequential change-point detection methods. *IEEE Transactions on Signal Processing*, 54(9):3372–3382, 2006.

[26] Matthias Neugschwandtner, Paolo Milani Comparetti, Gregoire Jacob, and Christopher Kruegel. Forecast: skimming off the malware cream. In *Annual Computer Security Applications Conference*, ACSAC, pages 11–20, ACM, 2011.

[27] Gregoire Jacob, Herve Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4:251–266, 2008.

[28] Mihai Christodorescu and Somesh Jha. Testing malware detectors. *SIGSOFT Software Engineering Notes*, 29:34–44, 2004.

[29] O. Sukwong, H. Kim, and J. Hoe. An empirical study of commercial antivirus software effectiveness. *Computer*, PP(99):1, 2010.

[30] Michael Venable, Andrew Walenstein, Matthew Hayes, Christopher Thompson, and Arun Lakhotia. Vilo: a shield in the malware variation battle. In *Virus Bulletin*, 2007.

[31] Bill Pollak. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006.

[32] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information System Security*, 3:186–205, 2000.

[33] Details about the discovered false alarms, appendix to the anti-vius comarative, september 2014. Technical report, AV Comparatives, 2014.

[34] D. Gao, M.K. Reiter, and D. Song. Beyond output voting: Detecting compromised replicas using hmm-based behavioral distance. *IEEE Transactions on Dependable and Secure Computing*, 6(2):96–110, 2009.

[35] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1-2):1–13, 2012.

[36] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference*, ACSAC, pages 421–430, 2007.

[37] Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *European conference on computer systems*, EuroSys, pages 61–74, ACM, 2009.

[38] Art Baker and Jerry Lozano. *The Windows 2000 Device Driver Book*. Prentice Hall, 2nd edition, 2000.

[39] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. In *IEEE Security and Privacy*, pages 120–128, 1996.

[40] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. Technical report, Symantec Corp., 2011.

[41] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. Duqu: Analysis, detection, and lessons learned. In *European Workshop on System Security*, EuroSec, ACM, 2012.

[42] Kate Munro. Deconstructing flame: the limitations of traditional defences. *Computer Fraud & Security*, 2012(10):8–11, 2012.

[43] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *International Symposium on Software Testing and Analysis*, ISSTA, pages 122–132, ACM, 2012.

[44] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2): 6:1–6:42, 2008.

[45] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[46] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. Technical report, Purdue University, 2007.

[47] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.

[48] M. Shafiq, Syed Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In *Detection of Intrusions and Malware, & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 88–107. Springer Berlin-Heidelberg, 2008.

[49] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, pages 38–49, 2001.

[50] Yanfang Ye, Tao Li, Qingshan Jiang, and Youyu Wang. Cimds: Adapting postprocessing techniques of associative classification for malware detection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(3):298–307, 2010.

[51] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Zero-day malware detection based on supervised learning algorithms of api call signatures. In *Australasian Data Mining Conference*, AusDM, pages 171–182, Australian Computer Society, Inc., 2011.

[52] Eitan Menahem, Asaf Shabtai, and Adi Levhar. Poster: Detecting malware through temporal function-based features. In *SIGSAC Conference on Computer & Communications Security*, CCS, pages 1379–1382, ACM, 2013.

[53] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. *IEEE Symposium on Security and Privacy*, 0:32–46, 2005.

[54] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, ESEC-FSE, pages 5–14, ACM, 2007.

[55] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7(4): 424–438, 2010.

[56] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *Security Symposium*, USENIX Association, 2006.

[57] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer Berlin-Heidelberg, 2006.

[58] Qinghua Zhang and D.S. Reeves. Metaaware: Identifying metamorphic malware. In *Annual Computer Security Applications Conference*, pages 411–420, 2007.

[59] S. Alam, R.N. Horspool, and I. Traore. Mard: A framework for metamorphic malware analysis and real-time detection. In *International Conference on Advanced Information Networking and Applications*, AINA, pages 480–489, IEEE, 2014.

[60] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, 30(5):1–54, 2008.

[61] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. pages 431–441, 2007.

[62] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Conference on computer and communications security*, CCS, pages 116–127, ACM, 2007.

[63] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Annual Technical Conference*, pages 18:1–18:14, USENIX Association, 2007.

[64] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):5:1–5:29, 2014.

[65] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *International Symposium on Recent Advances in Intrusion Detection*, RAID, pages 78–97, Springer-Verlag, 2008.

[66] Elizabeth Stinson and John C. Mitchell. Characterizing bots' remote control behavior. In *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA, pages 89–108, Springer-Verlag, 2007.

[67] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.

[68] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *SIGARCH Computer Architecture News*, 41(3):559–570, 2013.

[69] M.B. Bahador, M. Abadi, and A. Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *International eConference on Computer and Knowledge Engineering*, pages 703–708, 2014.

[70] Robert Moskovitch, Yuval Elovici, and Lior Rokach. Detection of unknown computer worms based on behavioral classification of the host. *Computational Statistics and Data Analysis*, 52: 4544–4566, 2008.

[71] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, pages 1–30, 2011.

[72] Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. Multi-channel change-point malware detection. In *Seventh International Conference on Software Security and Reliability*, SERE, 2013.

[73] Edward Stehle, Kevin Lynch, Maxim Shevertalov, Chris Rorres, and Spiros Mancoridis. On the use of computational geometry to detect software faults at runtime. In *International conference on autonomic computing*, ICAC, pages 109–118, ACM, 2010.

[74] Nong Ye, Xiangyang Li, Qiang Chen, Syed Masum Emran, and Mingming Xu. Probabilistic techniques for intrusion detection based on computer audit data. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 31(4):266–274, 2001.

[75] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Conference on Computer and Communications Security*, CCS, pages 399–412, ACM, 2010.

[76] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[77] Yihua Liao and V Rao Vemuri. Using text categorization techniques for intrusion detection. In *Security Symposium*, volume 12, USENIX Association, 2002.

[78] Dae-Ki Kang, D. Fuller, and V. Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *Annual Information Assurance Workshop*, pages 118–125, IEEE/SMC, 2005.

[79] Xin and Xu. Sequential anomaly detection based on temporal-difference learning: Principles, models and case studies. *Applied Soft Computing*, 10(3):859–867, 2010.

[80] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM, pages 15–26, ACM, 2011.

[81] A. Tokhtabayev, V. Skormin, and A. Dolgikh. Dynamic, resilient detection of complex malicious functionalities in the system call domain. In *Military Communications Conference (MILCOM)*, pages 1349–1356, 2010.

[82] B. Mehdi, F. Ahmed, S.A Khayyam, and M. Farooq. Towards a theory of generalizing system call representation for in-execution malware detection. In *International Conference on Communications*, ICC, pages 1–5, IEEE, 2010.

[83] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Security Symposium*, SSYM, pages 351–366, USENIX Association, 2009.

[84] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Leveraging string kernels for malware detection. In *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 206–219. Springer Berlin-Heidelberg, 2013.

[85] Han Xiao and Thomas Stibor. A supervised topic transition model for detecting malicious system call sequences. In *workshop on Knowledge discovery, modeling and simulation*, pages 23–30, 2011.

[86] Lakshmanan Nataraj, Vinod Yegneswaran, Phillip Porras, and Jian Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Workshop on security and artificial intelligence*, AISec, pages 21–30, ACM, 2011.

[87] Yang Zhong, H. Yamaki, and H. Takakura. A malware classification method based on similarity of function structure. In *International Symposium on Applications and the Internet*, SAINT, pages 256–261, IEEE/IPSJ, 2012.

[88] Kazuki Iwamoto and Katsumi Wasaki. Malware classification based on extracted api sequences using static analysis. In *Asian Internet Engineeering Conference*, AINTEC, pages 31–38, ACM, 2012.

[89] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Malwise: An effective and efficient classification system for packed and polymorphic malware. *IEEE Transactions on Computers*, 62(6):1193–1206, 2013.

[90] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW, pages 45:1–45:4, ACM, 2010.

[91] Younghee Park and Douglas Reeves. Deriving common malware behavior through graph clustering. In *Symposium on Information, Computer and Communications Security*, ASIACCS, pages 497–502, ACM, 2011.

[92] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware. *EICAR*, 2006.

[93] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security Privacy*, 5(2):32–39, 2007.

[94] J. Hegedus, Yoan Miche, A. Ilin, and A. Lendasse. Methodology for behavioral-based malware analysis and detection using random projections and k-nearest neighbors classifiers. In *International Conference on Computational Intelligence and Security*, CIS, pages 1016–1023, 2011.

[95] I. Firdausi, C. Lim, A. Erwin, and A.S. Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *International Conference on Advances in Computing, Control and Telecommunication Technologies*, ACT, pages 201–203, 2010.

[96] M. Apel, C. Bockermann, and M. Meier. Measuring similarity of malware behavior. In *Conference on Local Computer Networks (LCN)*, pages 891–898, IEEE, 2009.

[97] Saeed Nari and Ali A. Ghorbani. Automated malware classification based on network behavior. In *International Conference on Computing, Networking and Communications (ICNC)*, pages 642–647, 2013.

[98] N. Stakhanova, M. Couture, and A.A. Ghorbani. Exploring network-based malware classification. In *International Conference on Malicious and Unwanted Software (MALWARE)*, pages 14–20, 2011.

[99] D. Lobo, P. Watters, and Xinwen Wu. Rbacs: Rootkit behavioral analysis and classification system. In *Third International Conference on Knowledge Discovery and Data Mining*, pages 75–80, 2010.

[100] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, & Vulnerability Assessment*, DIMVA, pages 108–125, Springer-Verlag, 2008.

[101] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 2011.

[102] Tony Lee and Jigar J. Mody. Behavioral classification. In *European Institute for Computer Antivirus Research Annual Conference*, EICAR, 2006.

[103] Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification: bridging the static/dynamic gap. In *Workshop on Security and artificial intelligence*, AISec, pages 3–14, ACM, 2012.

[104] Senthilkumar G. Cheetancheri, John Mark Agosta, Denver H. Dash, Karl N. Levitt, Jeff Rowe, and Eve M. Schooler. A distributed host-based worm detection system. In *SIGCOMM workshop on Large-scale attack defense*, LSAD, pages 107–113, ACM, 2006.

[105] Haining Wang, Danlu Zhang, and K.G. Shin. Change-point monitoring for the detection of dos attacks. *IEEE Transactions on Dependable and Secure Computing*, 1(4):193–208, 2004.

[106] Alexander G. Tartakovsky, Boris L. Rozovskii, Rudolf B. Blazek, and Hongjoong Kim. Detection of intrusions in information systems by sequential change-point methods. *Statistical Methodology*, 3(3):252–293, 2006.

[107] George Forman. An extensive empirical study of feature selection metrics for text classification. *Journal of Machine Learning Research*, 3:1289–1305, 2003.

[108] William B Cavnar and John M Trenkle. N-gram-based text categorization. Technical report, Environmental Research Institute of Michigan, 1994.

[109] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREC*, volume 10, pages 1320–1326, 2010.

[110] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[111] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[112] Mark A. Hall Ian H. Witten, Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2011.

[113] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[114] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[115] Ashraf M. Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence*, volume 3339 of *Lecture Notes in Computer Science*, pages 488–499. Springer Berlin-Heidelberg, 2005.

[116] Harry L. VanTrees. *Detection, Estimation, and Modulation Theory.* John Wiley and Sons, 2001.

[117] A. Wald. *Sequential Analysis.* Wiley, New York, 1947.

[118] Pramod K. Varshney. *Distributed Detection and Data Fusion.* Springer, 1997.

[119] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *International Conference on Machine Learning*, ICML, pages 116–, ACM, 2004.

[120] Alexander Genkin, David D Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

[121] E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):pp. 100–115, 1954.

[122] Edda Leopold and Jörg Kindermann. Text categorization with support vector machines. how to represent texts in input space? *Machine Learning*, 46(1-3):423–444, 2002.

[123] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *Workshop on Artificial Intelligence and Security*, AISec, pages 99–110, ACM, 2013.

[124] Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004.

[125] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and Regression Trees.* Chapman and Hall/CRC, 1984.

[126] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[127] Eui-Hong (Sam) Han and George Karypis. Centroid-based document classification: Analysis and experimental results. In *Principles of Data Mining and Knowledge Discovery*, volume 1910 of *Lecture Notes in Computer Science*, pages 424–431. Springer Berlin-Heidelberg, 2000.

[128] Jean Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996.

Vita

Raymond J. Canzanese, Jr. was born in Lewisville, Texas on 4 December 1984. He is a U.S. citizen and current resident of Philadelphia, Pennsylvania. He attended Drexel University, obtaining his B.S. in Computer Engineering in 2008 and his Ph.D. in Electrical Engineering in 2015.

Publications

1. Raymond Canzanese, Spiros Mancoridis, and Moshe Kam. Multi-channel Change-Point Malware Detection. In *International Conference on Software Security and Reliability*, SERE, IEEE, 2013.
2. Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. Toward an Automatic, Online Behavioral Malware Classification System. In *International Conference on Self-Adaptive and Self-Organizing Systems*, SASO, IEEE, 2013.
3. Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. Inoculation against malware infection using kernel-level software sensors. In *International Conference on Autonomic Computing*, ICAC, ACM, 2011.