

IMPROVING HOST-BASED COMPUTER SECURITY USING SECURE ACTIVE MONITORING AND MEMORY ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Bryan D. Payne

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2010

Copyright © 2010 by Bryan D. Payne

IMPROVING HOST-BASED COMPUTER SECURITY USING SECURE ACTIVE MONITORING AND MEMORY ANALYSIS

Approved by:

Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Jonathon Giffin
School of Computer Science
Georgia Institute of Technology

Reiner Sailer
Global Security Analysis Lab
IBM T. J. Watson Research Center

Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Date Approved: 25 May 2010

In loving memory of my mother.

ACKNOWLEDGEMENTS

Most of life's meaningful accomplishments are only possible through the assistance of many other people. This dissertation and all of the related research are no exception. While it's impossible to enumerate everyone who contributed to this work, I would like to acknowledge some key contributors in this space.

The amazing people at the Georgia Tech Information Security Center (GTISC) clearly played a key role in this work. My advisor, Wenke Lee, and the rest of my committee members, Mustaque Ahamad, Jon Giffin, Karsten Schwan, and Reiner Sailer (from IBM Research), all provided thoughtful guidance throughout the entire lifecycle of this research. Many students in GTISC collaborated with me on this work including Martim Carbone, Brendan Dolan-Gavitt, Monirul Sharif, and Jinpeng Wei. Even more students including Kapil Singh, Manos Antonakakis, Abhinav Srivastava, Long Lu, and Artem Dinaburg were always happy to provide feedback, advice, proofreading, and any other help needed in the day to day life of a graduate student. Mike Hunter, Paul Royal, Michael Lee, Justin Bellmore, and Robert Edmonds also provided helpful guidance and support. And, finally, Mary Claire Thompson and Alfreda Barrow always managed to keep GTISC running smoothly regardless of the latest crisis. Thanks to everyone at GTISC for providing an environment that made this work possible.

Complementing the people in GTISC, my family and friends provided the encouragement and support that helped me keep my sanity throughout this process. Dave Roberts provided sound counsel throughout the ups and downs of my research. Rocky Dunlap and Dave Lillethun always asked the right questions and helped remind me of the importance of balance. Prior to starting at GTISC, Nick Petroni helped to inspire me to pursue systems security research.

My mother provided constant encouragement throughout my life and helped me to see the value of a good teacher. Her courageous fight against cancer showed me how to be graceful through adversity. For this and the many intangibles that a mother passes on to her children, I have dedicated this work to her. My father helped me see that this was possible as I watched him complete his dissertation by working long evenings when I was young. My brother has remained a steady voice throughout many struggles of his own and was always willing to help distract me from the stress of graduate school by talking about bicycles and cars.

The single largest influence in my life through graduate school has been my loving wife, Sara. She was always understanding when I needed to work late and always reminded me of why I went down this road in the first place. Without her love and support, none of this would have been possible. And her amazing food was always appreciated. Lastly, my son Sean has helped to re-spark my imagination and creativity as he discovers the world for the first time.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	x
LIST OF FIGURES	xi
LIST OF LISTINGS	xii
SUMMARY	xiii
 I SYSTEM SECURITY OPPORTUNITIES THROUGH VIRTUALIZATION	 1
1.1 Motivation	1
1.1.1 The Modern Era of Virtualization	1
1.1.2 Security Implications	2
1.1.3 External Monitoring	3
1.2 Challenges and Opportunities	3
1.3 Dissertation Overview	5
 II BACKGROUND AND RELATED WORK	 7
2.1 Host-Based Security	7
2.1.1 Secure System Design	9
2.2 External Host-Based Monitoring	18
2.2.1 Coprocessor-Based Monitoring	18
2.2.2 Virtual Machine-Based Monitoring	19
2.3 Virtualization	22
2.4 Memory Analysis Techniques	26
 III SECURE ACTIVE HOST-BASED MONITORING	 28
3.1 Motivation	28
3.2 Background Information	30
3.2.1 Previous Approaches	30

3.2.2	Xen Hypervisor	31
3.3	Security Requirements and Threat Model	34
3.3.1	Design Goals	34
3.3.2	Formal Requirements	36
3.3.3	Threat Model and Assumptions	39
3.4	The Turret Framework	41
3.4.1	Overview	41
3.4.2	Security VM Components	44
3.4.3	User VM Components	47
3.4.4	Hypervisor Components	49
3.5	Turret Implementation	51
3.5.1	Virtual Machine Introspection	53
3.5.2	Hardware Event Interposition	58
3.5.3	Hooks and Trampoline	60
3.5.4	Inter-VM Communication	61
3.5.5	Memory Protection	63
3.6	Security Evaluation and Analysis	66
3.6.1	Current Attacks	66
3.6.2	Future Attacks	68
3.7	Performance Evaluation	74
3.7.1	Passive Monitoring	75
3.7.2	Active Monitoring Using Hooks	77
3.8	Discussion and Future Work	80
3.8.1	Using Passive Monitoring	80
3.8.2	Using Active Monitoring	82
3.8.3	Future Work	83
3.9	Summary	84
IV	IMPROVING VMI'S RESILIENCE TO SOFTWARE CHANGES	86
4.1	Motivation	86

4.2	Previous Techniques	88
4.2.1	Hard-Coded Values	88
4.2.2	Heuristics	88
4.2.3	Debugger Tools	89
4.2.4	Code Analysis	89
4.3	Locating Data Structures Using Machine Learning	90
4.3.1	Preparing Data For Training	91
4.3.2	Selecting and Training the Classifier	93
4.3.3	Classifying Unknown Data	95
4.3.4	Using The Results	99
4.3.5	Accuracy and Performance Evaluation	100
4.4	Discussion and Future Work	105
4.5	Summary	106
V	GYRUS: A VMI-BASED SECURITY FRAMEWORK	107
5.1	Motivation	107
5.2	Previous Techniques	110
5.2.1	Identifying Human Intent	110
5.2.2	Detecting and Containing Malicious Activity	112
5.3	Requirements and Threat Model	113
5.3.1	Requirements	113
5.3.2	Threat Model and Assumptions	114
5.4	The Gyrus Framework	115
5.4.1	Hardware Event Interposition	117
5.4.2	Dynamic Authorization Creation	118
5.4.3	Enforcement	120
5.5	Gyrus Implementation	120
5.5.1	Hardware Event Interposition	121
5.5.2	Dynamic Authorization Creation	122
5.5.3	Enforcement	124

5.6	Summary	125
VI	APPLICATION CASE STUDIES USING GYRUS	126
6.1	Motivation	126
6.2	Application Case Studies	127
6.2.1	Email Case Study: Outlook Express	127
6.2.2	Web Browser Case Study: Internet Explorer	131
6.2.3	VoIP Case Study: Skype	137
6.3	Security Evaluation And Analysis	139
6.3.1	Current Attacks	140
6.3.2	Future Attacks	141
6.3.3	False Positive and False Negative Analysis	143
6.4	Performance Evaluation	144
6.5	Discussion and Future Work	147
6.6	Summary	150
VII	CONCLUSIONS	151
7.1	Summary and Contributions	151
7.2	Open Problems	154
7.3	Closing Remarks	157
	REFERENCES	158

LIST OF TABLES

1	HMM classification accuracy for location data structures in memory.	101
2	HMM classification accuracy for location data structures in memory, with post-processing steps included.	101
3	Outlook Express latency introduced by Gyrus for different user interac- tions. All times are in milliseconds.	144
4	Internet Explorer latency introduced by Gyrus for different user interac- tions. All times are in milliseconds.	144

LIST OF FIGURES

1	Formal model of secure active monitoring shown with potential attacks. . .	37
2	High-level view of the Turret framework and its core components.	42
3	XenAccess software architecture.	45
4	Steps needed to map a kernel memory page based on a kernel symbol using virtual memory introspection.	56
5	Data flow path for secure active monitoring.	62
6	Overview of the Turret page protection technique.	63
7	Achieving byte-level granularity using page-level protections.	65
8	Various forms of attacks aimed at circumventing the fine-grained active monitoring hooks installed by Turret in the User VM.	71
9	XenAccess memory mapping performance.	75
10	XenAccess memory read and write performance.	76
11	Micro-benchmarks showing performance of secure monitoring hooks. . . .	77
12	Overview of the steps we use for memory analysis.	90
13	Slicing memory data to create instances with a sliding window.	91
14	Evaluation steps for unknown instances.	95
15	Histogram motivating threshold value selection for HMM classification. . .	97
16	Data structures connected by a circular, doubly-linked list.	98
17	Formal requirements for a security framework driven by hardware events. .	113
18	Overview of the Gyrus framework.	116
19	High-level view of the Gyrus framework logic.	120
20	Windows user interface as reconstructed by Gyrus.	124
21	Gyrus modules supporting Outlook Express.	130
22	Gyrus logic for modules supporting Internet Explorer.	132
23	Software components involved in processing HTTP traffic with Gyrus. . . .	135
24	Time required for work performed by Gyrus when a user clicks on the send email button.	145

LIST OF LISTINGS

1	XenAccess example for listing all Linux kernel modules in the User VM. . .	81
2	Turret example code for receiving hardware events.	83

SUMMARY

Thirty years ago, research in designing operating systems to defeat malicious software was very popular. The primary technique was to design and implement a small security kernel that could provide security assurances to the rest of the system. However, as operating systems grew in size throughout the 1980's and 1990's, research into security kernels slowly waned. From a security perspective, the story was bleak. Providing security to one of these large operating systems typically required running software within that operating system. This weak security foundation made it relatively easy for attackers to subvert the entire system without detection.

The research presented in this thesis aims to reimagine how we design and deploy computer systems. We show that through careful use of virtualization technology, one can effectively isolate the security critical components in a system from malicious software. Furthermore, we can control this isolation to allow the security software a complete view to monitor the running system. This view includes all of the necessary information for implementing useful security applications including the system memory, storage, hardware events, and network traffic. In addition, we show how to perform both passive and active monitoring securely, using this new system architecture.

Security applications must be redesigned to work within this new monitoring architecture. The data acquired through our monitoring is typically very low-level and difficult to use directly. In this thesis, we describe work that helps bridge this semantic gap by locating data structures within the memory of a running virtual machine. We also describe work that shows a useful and novel security framework made possible through this new monitoring architecture. This framework correlates human interaction with the system to distinguish legitimate and malicious outgoing network traffic.

CHAPTER I

SYSTEM SECURITY OPPORTUNITIES THROUGH VIRTUALIZATION

1.1 Motivation

Computer security is in a state of crisis. State-of-the-art malware is now able to evade network security monitoring software [128]. Botnets are manipulating critical network services such as the domain name system (DNS) to eavesdrop on and steal information from users [46]. Companies supporting signature-based host security software (e.g., anti-virus) are unable to produce signatures as fast as new malware is created [122, 101]. And operating systems are routinely completely subverted by malware, making it nearly impossible to provide trusted execution environments for any security critical computations ranging from online banking to security software [131]. The average computer user has no meaningful way to respond to these threats because the fundamental building blocks needed to ensure system security are either broken or missing.

However, an opportunity now exists to provide these building blocks by reimagining system security. This thesis explores how this can be done through the proper use of modern virtualization technology. In particular, we demonstrate how to leverage virtualization to allow security software to run with a reduced risk of subversion, while still providing rich monitoring and response capabilities.

1.1.1 The Modern Era of Virtualization

Virtualization originally became popular in the 1960's and 1970's as a way to share a single expensive computer among many users. As personal computers grew in popularity through the 1980's, along with computer architectures that were more difficult to virtualize,

the popularity of virtualization waned. However, it has seen a resurgence over the past decade as desktop computers became powerful enough to run multiple operating systems concurrently. Virtualization has also seen a resurgence on the server side of the market where it is used to consolidate operating systems on fewer hardware resources, improve the manageability of large data centers, and enable new computing paradigms such as cloud computing. Hardware manufacturers have responded to this trend by creating a variety of virtualization-specific capabilities in recent processors, helping to ensure the longevity of this renewed interest in virtualization.

Identifying this paradigm shift towards virtualized computing, many companies and researchers are identifying new services that can leverage this new layer of abstraction. Techniques such as virtual machine migration, rapid operating system provisioning, improved backup capabilities, improved user desktop experiences (e.g., VMWare's Unity), and new software deployment techniques (e.g., using virtual machine appliances) are all enabled through virtualization. Virtualization has also enabled new business models such as cloud computing, where a user builds multiple virtual machines and pays to run them in a data center. Each of these developments has security implications specific to their design, however we must also consider the security implications of virtualization in general.

1.1.2 Security Implications

While many people have touted virtualization as the solution to today's computer security problem, closer inspection reveals that it is not quite that easy. In fact, when designed or deployed poorly, virtualization can negatively impact security. This is because using virtualization means adding more software to your computer, which means more opportunities for vulnerabilities. Furthermore, as data centers consolidate their servers, it is possible to now have multiple systems separated by software instead of an air gap. The end result could be quite damaging to security.

Fortunately, the picture is not all bad. When designed and deployed properly, virtualization can be very useful for both security and, a related topic, overall system management. Getting it right is not always trivial. As part of this thesis, we consider what it means to get it right, and suggest some strategies for getting the most security benefit from a virtualization platform. The key goal here is to ensure sufficient isolation between virtual machines to prevent malware from crossing this boundary.

1.1.3 External Monitoring

If we can ensure this isolation between virtual machines, then the next logical step is to use this isolation to separate untrusted software from security critical software. Perhaps the most security critical software on a system today is the software responsible for monitoring for security threats. This broad class of software, which includes anti-virus software and intrusion detection systems, is usually run within the same operating system that it is working to protect. However, this execution environment does not provide sufficient protections to stop malware from disabling or tampering with this security critical software.

Moving this software into its own virtual machine introduces new challenges. For example, we must balance the isolation needed to protect this software with the access needed for the tool to perform useful monitoring. In addition, we must identify the best way to allow the security software to access the information it needs, when it is needed. One of the primary contributions of this thesis is a framework that addresses these problems, enabling the development of robust security applications that can protect the operating systems running in multiple virtual machines while reducing the risk of malware subverting the security software.

1.2 Challenges and Opportunities

Moving security software outside of its original operating system and into an isolated virtual machine presents a variety of difficult research challenges. We discuss the most important high-level challenges below:

- *Accessing the information* that is needed for the software to operate. This information includes the system's memory, disk contents, network traffic, hardware events, and display buffer. In a traditional setting, the security software could access all of this information using the interfaces provided by the operating system. In this new setting, none of these interfaces exist so we must build both the interface and the mechanism for access to each piece of information that is needed.
- Knowing *when to access this information* is just as critical as being able to access it. When operating within the monitored operating system, security tools can easily place execution hooks throughout the system to receive notification of important events such as new process execution or file creation. In this new setting, there is no established way to perform active monitoring between virtual machines.
- Much of our access to information will be at a very low-level, so we need to bridge the *semantic gap* to extract useful information from these low-level bytes.
- *New types of attacks* are always a concern with a new architecture. For example, simply viewing data generated within the untrusted virtual machine introduces the possibility of data-driven buffer overflow attacks against the security tools. We must be careful to understand the security tradeoffs associated with this new architecture.
- *Performance* is always a concern for real-world systems. In this case, the security is only useful if the performance impact is acceptable to the users and administrators that will be working with the resulting systems.

The potential opportunities resulting from pursuing this new architecture for security software far outweigh the challenges described above. By protecting critical security software, we can restore trust in the fundamental building blocks for securing both server and desktop systems. This, in turn, limits the capabilities of today's malware while enabling a new generation of defensive techniques that can potentially combine more advanced host and network monitoring to defeat tomorrow's malware.

1.3 Dissertation Overview

This thesis investigates **a practical approach to enabling simple, flexible, and comprehensive active monitoring and memory analysis techniques for security software in a virtualized environment**. A *simple* approach is critical to the acceptance of this technology by allowing these techniques to be accessible to any programmer through a high-level API that provides a useful interface for security software. This interface must also be *flexible* by providing a variety of techniques for active monitoring and providing a common way to perform memory analysis for both live memory views and snapshots, as well as a common way to perform memory analysis on different operating systems. Finally, the interface must be *complete* to allow for active monitoring of any critical system event combined with passive monitoring of the entire system's memory. This completeness requirement provides security tools with an unobstructed system view while preventing malware authors from hiding behind the view provided by any particular API.

After discussing the related work in Chapter 2, we present our secure active monitoring framework in Chapter 3. This framework allows developers of security applications to perform event-driven, active monitoring by installing protected hooks in the monitored operating system, intercepting hardware events, and intercepting network traffic. It also provides passive monitoring capabilities, allowing developers to view the complete runtime memory state of the monitored operating system and all of its applications. We discuss the security and performance of this framework, and also provide small example applications to demonstrate how developers can utilize the framework.

In Chapter 4, we present a memory analysis technique for locating data structures within memory virtual address space. We demonstrate how this technique can work across various operating system versions, as opposed to previous work that relied on extensive reverse engineering of a specific operating system version.

In Chapter 5, we discuss a novel security application framework designed to distinguish user interactions from that of malicious software. Then, Chapter 6 demonstrates how this

can be used to create applications that fight spam and click fraud by correlating human generated hardware events with network traffic, using secure active monitoring and memory analysis.

Finally, this thesis concludes with Chapter 7 which describes the overall impact of this work and future opportunities in this research area.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Host-Based Security

Research in the area of host-based security has evolved somewhat drastically over the past forty years. This evolution can be traced to the penetration of computers into government, businesses, and private homes combined with the differing threats in these environments. In this section, we discuss the key achievements in host-based security as they relate to the research presented in the remainder of this thesis. However, in order to best understand the relations between each of the topics below, we first review the major turning points in computer security during this timeframe.

Much of the earliest work in computer security was driven by the military. This work was primarily motivated by the problem of having computers handle data of different classification levels. For this reason there was a focus on mandatory access controls and high assurance systems. The mandatory access controls provided a mechanism for enforcing a policy to protect classified information. And the high assurance systems provided confidence that the system was built correctly through a well documented engineering process.

By the 1980's, there was a growing interest in applying some of these security techniques to computers used in corporations. This spurred new research to understand how the security needs of businesses differed from that of the government. While the systems being built could largely be used in both settings, the security policies were quite different.

With the explosive growth of the personal computer in the late 1980's and early 1990's, there was a perception that security was not a critical feature for these machines. Unlike mainframe computers, these machines were often used by only one person and were either not connected to a network or only used a low bandwidth, intermittent connection (e.g.,

a modem). During this time commodity operating systems such as Microsoft Windows, Linux, and the Mac OS rapid grew in popularity. At the same time, security remained primarily the concern of the military and corporations running large multi-user machines.

As these commodity operating systems reached a critical mass around the mid-1990's, the Internet became available to the masses, forever changing the face of computer security. Malicious software, previously a minor concern as it was restricted to spreading by floppy disk, had a new propagation vector. Initially, malicious software designed to propagate through the Internet was authored by hobbyists seeking bragging rights. However, it was not long before someone realized that large groups of compromised computers could be used to send unsolicited email advertisements (i.e., SPAM), while avoiding the detection of the major network operators. With this strong profit motive, malicious software quickly transitioned from the domain of the hobbyist to the domain of professional criminals. The level of sophistication, along with the quality of code, for these new generations of malicious software increased considerably. Compromised computers have themselves turned into a commodity such that attackers will now go to great lengths to avoid detection by anti-virus software (e.g., using polymorphic code) and by the humans interacting with the machine (e.g., throttling network utilization and even fixing system configuration problems).

It was during this period that research in computer security ramped up significantly. Looking beyond the idea of building a secure operating system, researchers began exploring how to detect and prevent malicious software from entering the commodity operating systems found on the vast majority of modern computers. People explored techniques for intrusion detection systems, malicious software analysis, and for securing the larger computer network infrastructure, along with many other related topics.

During this most recent period, it no longer made economic sense for the military to develop their own operating system. Many of the government-sponsored projects of the 1970's and 1980's to create secure operating systems were finished and would likely not

be revisited. Instead, the government switched to using commodity operating systems. However, these systems didn't offer the mandatory access controls or the high assurance previously required by the military. Instead, the military has sought ways to harden these operating systems, often with mixed results.

Today, host-based security research continues working to address the challenges of preventing, detecting, and analyzing malicious software on commodity systems. However, with the rapidly evolving threat, researchers have returned to many of the fundamental techniques first used in the 1970's in an effort to provide a stronger foundation to the security in commodity operating systems. The work within this thesis is one example of this, as we explore techniques for moving trusted security applications out of the user's operating system, and into a more protected environment. As described below, this can be viewed as a modern day interpretation of the separation kernel concept first introduced in the early 1980's.

2.1.1 Secure System Design

The primary goal of most secure system designs is to build computer systems that can prevent security breaches, or at least mitigate any damages caused by such breaches. This goal has remained unchanged for nearly fifty years. In this section we look at the steps taken by various systems to achieve this goal throughout the years. We start by looking at Multics, an operating system whose design dates back to the mid-1960's. We then review work on the use of security kernels, separation kernels, and virtualization in the 1970's and 1980's. Next we consider the influences from both the government and commercial sectors during the rapid growth periods of the 1980's and 1990's. Finally, we look at how this history has shaped current approaches to secure systems design over the past decade.

Multics Multics is an operating system that began in 1965 and was used through 2000 [171]. The original Multics design was presented in six papers at the Fall Joint Computer Conference in 1965 [43, 72, 175, 47, 123, 48]. Security was one of the original design goals

for Multics, resulting in various features that made it more secure than many of today's more modern systems. These features included programming the system in PL/I (a higher level language that was effectively immune to buffer overflows), using hardware security features such as the "no execute" bit that prevents execution of data regions, and designing the kernel to reduce complexity. This attention to security made Multics the most secure operating system of its time, however a security evaluation by Karger et al. [94] in 1974 revealed a variety of potential problems. The evaluators suggested that Multics be enhanced using mandatory access controls and a security kernel to combat potential problems with malicious software such as trojan horses. While Multics did eventually achieve a Class B2 security evaluation from the National Computer Security Center, efforts aimed at further improving its security were eventually terminated [95].

Security Kernels As interest grew in designing secure systems, people began thinking about the best design practices for these systems. Anderson wrote a two volume report that detailed some of these best practices [6, 7]. This report presented a reference monitor as an entity that enforces all access relationships between all subjects and objects of a system. An implementation of this reference monitor concept is called a reference validation mechanism, and some of the earliest actual implementations took the form of a security kernel. A security kernel is a subset of the operating system that performs reference validation, and is small enough to be verified and audited. A security kernel runs as the most privileged code in the system.

Security kernels quickly became a popular implementation technique for secure system design that was used in a variety of systems including the PDP-11/45 [150], the Kernelized Secure Operating System (KSOS) [62], the Gemini Multiprocessing Secure Operating System (GEMSOS) [60], the Honeywell Secure Communications Processor (Scomp) [64], and the Logical Coprocessing Kernel (LOCK) [149] based on the Provably Secure Operating System (PSOS) work by Feiertag and Neumann [61]. Many of these designs proved to be

at least partially successful from a security viewpoint. For example, the GEMSOS system achieved an A1 rating – the highest possible – under the Trusted Computer System Evaluation Criteria (TCSEC) [51]. However, as Ames noted in 1981 [5], security kernels are not a panacea to the problems of secure system design. In particular, Ames described that the reference monitor concept is too simple to fully model all of the interactions needed to secure real multi-level systems. Furthermore, security kernel-based systems would usually make use of trusted processes as a way to perform trusted functionality outside of the security kernel itself. While these trusted processes were supposed to be small in both number and size, and undergo the same security analysis as the security kernel itself, these restrictions were typically not imposed in practice.

Separation Kernels and Virtualization Responding to the deployment concerns associated with security kernels, Rushby proposed a slightly different approach called a separation kernel [142]. Rushby believed that a single secure system could be reimaged as a distributed system, where security is partly achieved through isolation between each system provided by a specialized security kernel that he called a separation kernel. Using this architecture, the trusted processes could run isolated from the other parts of the system, simplifying both verification and deployment. Rushby showed formally that a separation kernel could provide the isolation needed to achieve these goals [141].

System-level virtualization, the technique of running multiple operating systems on a single hardware platform, is one way to implement a separation kernel. Virtualization also has the security benefit of reducing the probability of failure for the trusted processes, as argued by Madnick and Donovan [110]. For these reasons, virtualization has proven to be a useful architectural component in secure system design for many years. In a related effort, Karger et al. designed and implemented the VAX VMM security kernel [96] which was a complete virtualization platform built to achieve an A1 rating under the TCSEC. While this effort was more general than a separation kernel because it was not strictly designed

to run a secure system decomposed into separate components, it could have been used for this purpose.

More recently, two projects have implemented the separation kernel concept to create high assurance systems, with a focus on embedded systems. The Multiple Independent Levels of Security (MILS) architecture uses a layered approach with a separation kernel, implemented as a micro-kernel, that runs multiple partitions, each running at one or more security levels [4]. MILS controls the isolation between each partition by a policy that defines data isolation, information flow control, periods processing, and damage limitations for the entire system. The Mathematically Analyzed Separation Kernel (MASK) project had similar goals, using a smart card as a target platform [111].

This thesis builds most directly on the idea of a separation kernel. In particular, we consider the need to have trusted processes that can monitor other parts of the system in order to ensure code and data integrity while enforcing other security policies. The separation kernel systems discussed above, and even modern virtualization platforms in general, do not provide adequate capabilities to support monitoring applications. This thesis addresses this problem, showing how to securely integrate both active and passive monitoring techniques into modern virtualization platforms. In the long term, we envision building on this work to support similar capabilities for more robust and verifiable separation kernels.

Security Evaluations With a growing number of computing systems (i.e., operating systems, databases, application software, networking components) under development in the mid 1980's, the government wanted an evaluation procedure in place to compare the relative security offered by each of these systems. This resulted in a series of evaluation standards, informally known as the “rainbow series” after the array of colors used for the each publication in the series. The primary book in this series was the Trusted Computer System Evaluation Criteria (TCSEC) [51], also known as the orange book. This book provided guidelines for evaluating a system and categorizing it in a particular division.

Divisions included D (all tested systems that failed to meet the criteria of another level), C (discretionary access control), B (mandatory access control), and A (formally verified). Each division contained one or more classes, further distinguishing the software into ratings such as A1, C2, etc. Subsequent books in the rainbow series provided additional guidance to evaluators regarding how to evaluate systems. For example, the red book provided the Trusted Network Interpretation [120] and the purple book provided the Trusted Database Management System Interpretation [121].

By the 1990's several countries each had their own security evaluation process. The United States used TCSEC, as described above. Canada used the Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) [12] and several European countries used the Information Technology Security Evaluation Criteria (ITSEC) [40]. Having different standards in different countries created confusion and difficulties for the vendors, so a new international standard called the Common Criteria for Information Technology Security Evaluation (Common Criteria or CC) [42] was developed to replace each of these previous evaluation processes. The Common Criteria is a very flexible evaluation framework that permits the vendors to specify precisely what security properties they aimed to achieve and how rigorous of an evaluation is desired. Successful evaluations result in the software achieving an Evaluation Assurance Level (EAL) ranging from 1 to 7, where 7 indicates the most thorough evaluation.

All of these security evaluation processes are designed to provide some level of assurance in the software development. Software assurance is defined as the “level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle, and that the software functions in the intended manner” [41]. Assurance is largely centered around the process of making software. Therefore, the work presented in this thesis on an architecture for secure active monitoring and techniques for performing memory analysis on running systems could be

implemented at various assurance levels as required based on the intended use of the resulting software.

Commercial Growth During the 1980's and 1990's, the commercial popularity of the desktop computer soared. This popularity was largely fueled by less expensive hardware combined with more user friendly operating systems such as Mac OS and Windows. This shift is a notable turning point with regards to systems security because, unlike the military-driven operating system development projects described above, these operating systems were not designed with security as a primary goal. However, as the popularity of these systems grew, it became increasingly difficult for the government to justify the expense of building and maintaining their own operating systems. Today, combining the price of these operating systems with the availability of many applications, many people have argued that it is no longer practical to build another operating system from the ground up. As a result, by the late 1990's systems security research shifted focus from designing a new secure operating system, to improving the security of existing commodity operating systems such as Windows.

Microsoft successfully achieved an EAL4+ rating for several versions of Windows, however this was done using a narrow set of security properties. This testing also required Windows to be configured differently than how it is normally deployed in homes and businesses today. For these reasons, many people argue that the EAL4+ certification for Windows is not very meaningful. Similar certifications exist for Linux and Solaris.

In an effort to improve the security of commodity operating systems, some projects have focused on adding security extensions to these existing systems. One such effort is the Security Enhanced Linux (SELinux) project, maintained by the National Security Agency (NSA). SELinux integrates with existing Linux operating systems to provide mandatory access control that enables type enforcement [13], role-based access control, and multi-level security policies. SELinux is known for being highly flexible, which has the drawback

of requiring complex policies. Even so, it is now deployed with popular Linux distributions such as Red Hat, albeit with a somewhat weak default policy. Sun created Trusted Solaris, offering similar functionality as SELinux. Today, Trusted Solaris has evolved into the Trusted Extensions that are available with Open Solaris. Efforts such as SELinux and Solaris Trusted Extensions do provide a useful additional layer of security on top of these commodity systems. The primary critiques of these systems are the highly complex policies and the fact that it is hard to build security on top of an insecure system.

Nevertheless, we believe that additional research in this area is valuable and that it would integrate well with the work presented in this thesis. For example, we completed some preliminary research that demonstrated how layered mandatory access control policies could help to reduce their complexity [127]. A natural way to perform this layering is to follow the existing layers in the system: the hypervisor, the operating system running in each virtual machine, and the applications running in each operating system. Security policies such as these (i.e., access control policies, integrity policies, etc.) are useful security tools that are orthogonal to the security monitoring research presented in this thesis. We envision future systems where the monitoring components can verify that such policies are working properly, or even serve as a mechanism to implement such policies in the case of active monitoring.

The Modern Era Most recently, secure systems research has continued to work under the assumption that users will be running a commodity operating system. Therefore, current research has focused on many of the boundary issues associated with ensuring that these systems remain secure from the initial boot until they are shut down. In addition, some researchers have even gone so far as to attempt to remove the operating system from playing a role in application-level security [34]. There is significantly more work in this space than can be reviewed here. Entire books have been devoted to the topic [71, 86]. Instead, in this section we provide an overview of the work that is most relevant to this thesis. The topics

presented here are either applications that could benefit from the work presented in this thesis, or complementary security work that could be used in conjunction with our work.

A challenging and often overlooked security problem is that of secure boot. Simply stated, secure boot is a technique to provide assurance that the software loaded after you turn on a computer is what you intended (e.g., a copy of Windows without any malicious software). Arbaugh et al. addressed this problem by verifying each stage of the boot process before it executes [9]. This ensures that the software is as intended at the time that it *starts* executing, but once execution proceeds one needs other techniques to validate the software integrity at runtime.

Various techniques have been proposed for verifying system-level software integrity at runtime. Sailer et al. use the trusted platform module, a trusted hardware component, to perform integrity measurements on all software from the BIOS up to the applications [147]. Others have used a hypervisor to run trusted code that monitors or enforces software integrity. SecVisor, for example, is a hypervisor that prevents changes to the kernel code by interposing on transitions between user and kernel space [155]. And Petroni et al. used a hypervisor to provide a secure monitoring environment from which to verify semantic integrity constraints in a running kernel [130].

Researches have also explored techniques for application-level software integrity verification. Abadi et al. introduced a technique known as control flow integrity (CFI) that validates a program's control flow graph against a known good version [3]. Later, Petroni et al. introduced state based control flow integrity (SBCFI) which effectively extended this concept to work on operating system kernels using an external monitor such as the monitoring framework presented in this thesis [131]. Building on this idea, Baliga et al. developed the Gibraltar system to automatically infer invariants on both control and non-control kernel-level data [15] using a modified version of Daikon [59]. Gao et al. came up with a different approach known as behavioral distance in which the same input is passed to multiple applications that are each evaluated based on how they process the input [67]. For

example, a potentially malicious input could be sent to several different web servers to see if it exploits any of them, causing their resulting behavior to deviate from the other servers.

Even when applications are behaving properly, a secure system requires a policy to validate its information-flow integrity. Checking for this can help prevent security issues such as allowing a system user to overwrite a critical password or configuration file. The Clark-Wilson model was among the first to formally look at data integrity policies [37]. However, Clark-Wilson was somewhat limited in terms of practical deployment due to the formal verification required for each application. Shankar et al. addressed this problem by weakening the formal verification requirements and providing a technique to automate the verification that was still needed while still enforcing a meaningful information-flow integrity policy known as CW-Lite.

Orthogonal to the work on software integrity, many researchers have leveraged virtualization to design innovative security architectures. NetTop is a Department of Defense project aimed at deploying operating systems that can process different classification levels on the same physical hardware [113]. This was done using a different virtual machine for each operating system, and a careful assessment of the communication channels (both covert and overt) between the virtual machines. The Terra architecture by Garfinkel et al. uses virtualization to create a “closed box abstraction” that permitted execution of a third party virtual machine without the risks of information disclosure or tampering [68]. McCune et al. developed the Shamon architecture which used virtualization on multiple physical machines to create a shared reference monitor for controlling mandatory access control across multiple machines [112]. And, more recently, several different groups of researchers created systems that used context sensitive page mappings [139] to protect application memory from the operating system that it runs on [184, 34]. These special page mappings provide a different view of memory based on who is accessing the memory (kernel or application) or how it is being accessed (read, write, or execute). Using this system,

applications received a plaintext view of their memory while the kernel receives an encrypted view. The mediation is performed by a trusted hypervisor.

2.2 External Host-Based Monitoring

External monitoring is when the monitoring software is securely separated from the source of the data being monitored. For network monitoring, this is relatively simple. Network monitors can view network traffic at any point on the network, whereas the traffic is created only at the hosts. This makes it very difficult for an attacker to compromise network monitoring software. However, with host based monitoring, it is more difficult to achieve this separation. Viewing events happening within an operating system has traditionally required running software within that operating system. Yet running software within the operating system makes it vulnerable to any malicious software running on the same system. For this reason, coinciding with the shift to less secure commodity operating systems discussed above, it has become increasingly difficult to perform secure host-based monitoring.

For these reasons, over the past decade researchers have explored ways to perform external host-based monitoring. The primary challenge is to obtain all of the information that one could get locally, while benefiting from some form of isolation. The two key techniques that have been explored for secure external host based monitoring include coprocessor-based monitoring and virtual machine-based monitoring.

2.2.1 Coprocessor-Based Monitoring

Noting the growing trend toward symmetric multi-processor (SMP) systems in 2000 (i.e., computers with two equally powerful processors), Hollingworth and Redmond developed a system that utilized one processor for “oversight” and another for “applications” [85]. The goal of this system was to provide greater protection for security monitoring software running on the oversight processor. Their architecture is effectively an early form of external monitoring. Today’s systems running multiple processor cores that are multiplexed using a hypervisor have many similarities to this earlier work.

Unfortunately, SMP systems did not continue to gain much popularity at the time of Hollingworth and Redmond's work. It wasn't until several years later, with multi-core processors, that desktop parallelism began to take hold. In the mean time, researchers discovered that security monitoring can achieve isolation by running on a coprocessor PCI card. Such cards were already available to the general public, but were still somewhat expensive because it was effectively an entire computer on a single PCI card. Researchers ran their own operating system on this coprocessor card, and showed that the host computer could be monitored through the PCI bus, allowing for a wide variety of security applications. This idea was first proposed in 2002 by Zhang et al. [188]. Two years later, Petroni et al. provided a more in-depth treatment and details about how the techniques could be used to monitor runtime kernel integrity [129].

Coprocessor based monitoring was a great step forward in that we could now do host monitoring while providing some protection for the security application. However, it also had some rather serious limitations. Coprocessor cards are expensive, and their view into the host is somewhat limited. To address these issues, researchers began exploring software-based solutions to the problem.

2.2.2 Virtual Machine-Based Monitoring

Virtualization is the primary technique used by researchers today to isolate security applications from potentially malicious execution environments. A typical configuration for both desktop and server environments starts with a hypervisor; a small software layer running directly on top of the hardware. All of the security-relevant applications are deployed in one virtual machine, whereas the user's desktop applications or the server components are deployed in one or more additional virtual machines.

Virtual Machine Introspection Virtual machine introspection (VMI) is a technique where processes in one virtual machine can view the runtime state of another virtual machine. This term was coined by Garfinkel and Rosenblum in 2003 when they introduced Livewire,

a host-based intrusion detection system based on VMI [69]. However, the idea of monitoring the runtime state of virtual machines dates back to 1973 with the Virtual Hardware Monitor by Casarosa and Paoli [31]. The runtime state available through VMI typically includes complete read and write access to memory, read access to processor registers, and some meta-data about the virtual machine. In some cases, data going into and out of the virtual machine such as network traffic or hard disk storage are also available.

The key drawbacks to VMI are performance and the semantic gap problem. Performance is a concern with some applications because of the time required to map memory pages across virtual machines. Monitoring activities that require a large number of page mappings are most affected by this problem. The semantic gap problem is the problem of extracting meaningful information from the low-level memory view typically provided by VMI. Research in the area of memory analysis is working to address this problem, as discussed later in this chapter.

Virtual machine introspection has proven to be a useful building block for the development of monitoring applications that desire the isolation benefits of running in a separate virtual machine. While these benefits have been most commonly associated with security applications, any systems monitoring application could be built using VMI. The primary contributions of this thesis are related to improving VMI through the addition of event driven monitoring techniques, and building useful security applications that utilize these techniques.

Runtime Monitoring Runtime monitoring applications use VMI to analyze and validate the state of a system in near real time. Ideally, attacks should be detected quickly enough to allow for defensive action to stop the attack and mitigate any side effects. The original VMI application, Livewire, described techniques to compare kernel and user views of the system, perform application-level integrity checking, detect the use of raw sockets, detect invalid memory writes, and detect when the network card is placed in promiscuous mode

[69]. Subsequent research built on these ideas and applied VMI to a variety of security applications.

Many of these applications performed a variety of monitoring tasks to create intrusion detection systems. HyperSpector used VMI to develop an intrusion detection system for a distributed environment based on capturing network traffic, mapping process address spaces, and viewing the target system's disks [103]. IntroVirt created vulnerability-specific predicates that could then be used to identify intrusions at runtime or by looking at historical system snapshots [91]. The VMwatcher system has similar goals but focuses on the semantic gap portion of the problem [88]. Xu et al. developed a usage control framework for kernel integrity protection that runs using VMI [183]. Psycho-Virt is another VMI based intrusion detection system that combines many of the above techniques [14]. VICI focuses on repair techniques that can help systems recover from attack using VMI [65]. Finally, Litty's master thesis provides an in-depth discussion of the design, creation, and testing of a VMI-based intrusion detection system [108].

Another application, related to intrusion detection, is process tracking and detection. Antfarm operates from within the hypervisor, inferring process existence by detecting address space identifiers such as the CR3 register value on the x86 architecture [89]. Similar techniques could be implemented using VMI. The same authors also developed Lycosid to detect hidden processes using cross-view validation [90].

VMI is also a natural fit for monitoring honeypots, many of which are already running in a virtual environment. Asrigo et al. developed an intrusion detection system for honeypots using VMI [10]. They noted that VMI allows for good visibility, which in turn meant that they could successfully detect intrusions in their honeypot with a small number of sensors. In related work, Jiang and Wang created the VMscope system to perform external monitoring of high-interaction honeypots using VMI [87]. VMscope could be hidden from an attacker like previous external monitoring techniques, but could also provide semantically rich information such as system call events, due to their use of VMI.

Forensic Monitoring Forensic monitoring and analysis occurs after a system is known to have been attacked. Instead of detecting or preventing an attack, the goals in this case are to learn more about what happened during the attack. VMI is typically used to look at the memory of a live, running system. However, one can also use it to simply take snapshots of the system state while it is running. These snapshots can then be analyzed after an intrusion occurs to determine details about the intrusion. Using VMI for forensic analysis provides the same benefits of cross-view-diff approaches, such as Strider GhostBuster [179], in that you can view the system from different semantic levels to identify inconsistencies. However, VMI has the added benefit that it can operate at runtime without restarting the system.

ReVirt uses snapshots obtained through VMI to enable instruction-level replay of the system, allowing for detailed and arbitrary forensic analysis [56]. BackTracker [98, 99] is one example of the type of analysis that could be performed on these snapshots. BackTracker can work backwards from an attack’s detection point to determine the point of entry. It can even reconstruct a graph showing the sequence of events that transpired during the attack. In related work, Hay and Nance created the VIX tools to perform forensic analysis of virtual machines running on Xen [80]. They described VMI as being very useful for forensics use given the fact that one can monitor the system non-intrusively.

2.3 *Virtualization*

The idea of virtualization was originally conceived in late 1964 by Creasy and Comeau from IBM as a way to partition the resources of mainframe computers [169]. Initially, and even today, virtualization was simply used to create the abstraction of multiple virtual machines running on a single hardware platform. However, more recently virtualization has been used as a supporting infrastructure for a variety of applications, including security. We take this approach in this thesis, although we do so cautiously noting that off-the-shelf virtualization solutions likely do not satisfy the requirements for a robust security architecture. Custom solutions will be required for deployable security solutions in most

cases. However, for the work in this thesis, we recognize that off-the-shelf virtualization can be used to demonstrate the viability of key technologies before investing in expensive, custom virtualization solutions.

In this section, we review previous work in virtualization as it relates to its suitability for supporting security architectures and applications. We start with a brief history of virtualization, then look at how it has been used for isolation purposes, and finally consider the question of virtualization’s suitability for security applications directly.

A Brief History The earliest implementations of virtualized systems were built in the late 1960’s and early 1970’s [169]. These systems were largely motivated by the desire to allow multiple users to work on the same hardware, while allowing each user direct access to the hardware for applications such as operating system debugging and development, running diagnostic software, or running a different operating system [74]. By 1974, there were at least ten virtual machine systems either available for use or under development including the VM/370, CP-40, CP-67, 360/30, and the PDP-10 [73]. With the shift to x86 hardware – which is harder to virtualize – for both server and desktop machines, virtualization fell out of favor during most of the 1980’s and 1990’s.

By 1998, VMWare had solved many of the challenges associated with virtualizing the x86 architecture. Soon after, they released both desktop and server class virtualization products that effectively renewed interest in virtualization technologies [172]. In the following years, virtualization became increasingly ubiquitous, in part because of open source projects such as Xen [16] and KVM [134]. While many people currently use virtualization products for the same reasons cited in the 1970’s (e.g., running multiple operating systems or improved hardware utilization), a new trend has emerged with people looking at ways to creatively use virtualization as part of a comprehensive rethinking of host-based system architectures. Virtualization is now used to support a wide variety of services including migration, fault tolerance, debugging, host monitoring, and sand boxing. Whitaker

et al. designed an extensible virtual machine monitor for supporting these types of services [182]. An ongoing debate questions if these applications of virtualization are actually turning virtualized systems into a modernized form of microkernel [79, 82]. Regardless of the answer, this debate illustrates that virtualization is now used to support a wide variety of applications.

Isolation Isolation is one of the key virtualization properties used to motivate its use for security applications and architectures. Traditionally, virtualization provides complete isolation between virtual machines. In this case, virtual machines have no more or less connectivity to each other than to non-virtualized hosts. Madnick and Donovan were the first to recognize the security benefits of this in 1973, suggesting that the isolation between virtual machines would provide stronger security than the isolation between processes within a given operating system [110]. Around the same time, Lampson defined a set of rules for confining an arbitrary program [104]. Drawing on the work from Madnick and Donovan, these rules would be much easier to enforce using virtualization instead of a traditional operating system. Similarly, Saltzer and Schroeder discussed the concept of controlled sharing noting that it is a simple concept that is difficult to implement given the mechanisms required [148], but this can also be simplified using virtualization.

Many researchers have leveraged this isolation property to make stronger statements about the security of their systems. Rushby proposed a separation kernel as a small kernel that provides strong isolation between the various processes that it runs [141]. This separation kernel concept was implemented using virtualization by Kelem and Feiertag [97]. Shockley and Schell proposed used “TCB subsets” to simplify the process of evaluating the security of large systems [158]. These subsets must be isolated from each other, only communicating through well defined channels, which is easier to implement using the virtualization abstraction.

Modern virtualization platforms provide for a significant amount of sharing between

virtual machines. One technique to control this sharing, and to enable the types of isolation applications discussed above, is to implement a mandatory access control policy within the hypervisor. Karger discussed the requirements for this approach in 2005 [93]. Around the same time, Sailer et al. implemented mandatory access control for Xen [146]. These mechanisms provide the foundation for controlling interactions between virtual machines, however identifying the techniques and policies required to achieve these goals remains an active research area [140].

Suitability In recent years, some researchers have questioned the suitability of using virtualization to address security problems. For example, Bellovin expressed concern over the potentially vulnerable interfaces required for interaction between virtual machines and the higher administrative burden imposed by virtualization [22]. And Garfinkel and Rosenblum identified several ways in which the rapid provisioning of virtual machines can complicate security administration [70]. All of these concerns are valid, but they are only critical of specific virtualization use cases.

The architecture proposed in this thesis does not correspond to these use cases. Instead, we use virtualization strictly as a technique to provide controlled isolation between the security-critical software and the user or server operating environment. This use case leverages previous work on isolation in a virtualized environment, while also benefiting from the application flexibility afforded by virtualization [33]. Using a custom hypervisor could further improve on this deployment scenario by reducing the size of the trusted computing base (TCB) [84]. Finally, the major challenges for implementing a secure hypervisor on the x86 platform [138] are no longer a concern due to the recent virtualization extensions added by both Intel and AMD to the base x86 architecture. For these reasons, we believe that virtualization is well suited for supporting the security architecture proposed in this thesis.

2.4 Memory Analysis Techniques

As discussed in Section 2.2, virtual machine introspection (VMI) is a technique that enables one virtual machine to view the runtime state of another virtual machine. The most common runtime state information accessed using VMI is memory because it contains a significant amount of information about the current state of the system. The major drawback to this approach is that it can be difficult to extract meaningful and useful information from raw memory. This is because the memory view provided through VMI is very low-level and it lacks any semantic information. The techniques used to extract relevant information from a raw memory view are collectively known as memory analysis.

The field of memory analysis first became popular within the digital forensics community. This community realized that one could acquire a memory snapshot of a system and then use that snapshot to extract forensic evidence detailing the activities happening on the machine at the time of the snapshot. Earlier work in this space, dating back to 1999, focused on techniques to scan memory, looking for malware signatures [164]. By 2005, forensic analysts began reconstructing the virtual address space and extracting the memory associated with the kernel and specific processes running on the system [28, 29]. The next major step forward was the introduction of FATKit, a framework for memory analysis created by Petroni et al. [132]. FATKit could extract the information needed for analyzing kernel data structures through static analysis of the kernel source code while providing a generic and extensible framework for performing memory analysis. While FATKit was never made publicly available, one of the same authors eventually released Volatility, which provided many of the same features, as an open source project [177].

Following the release of Volatility, many people in the forensics community discovered ways to extract a wide variety of useful information from memory. Schuster showed how to locate processes and threads [152] and how to extract data from pool allocations [151] in Windows. Kornblum showed how to interpret Windows page table entries to identify

swapped and non-initialized memory regions [102]. Dolan-Gavitt described how to extract more semantically meaningful data from windows processes using their associated virtual address descriptor (VAD) tree structure [52]. He also described techniques for extracting and analyzing the Windows registry from memory [53]. And Arasteh and Debbabi showed how to extract a process' execution history by analyzing its stack [8]. All of these techniques were designed to be very practical and most included demonstration code to show how the technique works. While these techniques are extremely useful, their primary drawback is that they only work on specific versions of Windows. Porting the techniques to work on other versions of Windows, or other operating systems, would require considerable effort.

Security tools deployed using VMI must do some memory analysis to gather runtime state information from the system. As VMI has grown in popularity, researchers have begun searching for ways to make this memory analysis more general and more robust. Cozzie et al. leveraged the inherent structure of program data to identify data in memory, and parse it based on how it is referenced by other code and data [44]. And Dolan-Gavitt used a technique similar to fuzzing to identify the portions of data structures that are required for system operation, and therefore that are most useful for creating data structure signatures [54].

In Chapter 4, this thesis explores general techniques for locating data structures in memory across a wide variety of operating system versions.

CHAPTER III

SECURE ACTIVE HOST-BASED MONITORING

3.1 *Motivation*

As malware has become increasingly sophisticated over the past several years, it is no longer unusual to see it disable critical security services on a victim's machine. Researchers responded to this threat by moving security services into different virtual machines (VMs) [69]. Techniques such as virtual machine introspection (VMI) make this possible by bridging the semantic gap between the Security VM and other VMs running on the same platform. In particular, VMI and related techniques have been used to build a wide range of security tools including intrusion detection systems, memory and disk integrity checkers, and system monitors [103, 125, 89]. All of these tools share one thing in common: each relies on *passive monitoring*. Passive monitoring is when the security tool monitors by external scanning or polling. As a result, it is unable to guarantee interposition on events before they happen.

This fundamental limitation of passive monitoring means that it is not a sufficient technique for implementing a full-featured anti-virus, intrusion detection, or intrusion prevention system. Previous efforts to implement these types of systems within a Security VM have resorted to implementing the systems with crippled functionality. What was missing in these systems was the ability to do *active monitoring*. Active monitoring is when the security tool is actively notified of system events when they happen, instead of polling to find out after the fact. When a critical system event occurs (e.g., execution reaching a hook or a particular hardware event), it will interrupt execution and pass control to the security tool. Active monitoring can be done both inside and outside of the system being monitored. Each location offers tradeoffs in security and visibility.

In order to achieve the best visibility, active monitoring should be performed by placing hooks inside the untrusted User VM. However, this type of active monitoring is challenging in the types of virtualized security architectures used in recent research. The problem is that it requires security-critical code inside untrusted User VMs. Since a major reason for moving to a virtualized architecture is to remove security-critical code from the untrusted User VMs, this feels like a step backwards. Properly protecting this code is sufficiently challenging that some researchers have attempted to avoid the problem altogether [88], resulting in systems that can only detect attacks, not prevent them. Furthermore, recent work has focused on providing strong protections for the entire kernel code [155], but these do not protect kernel data so they are insufficient for protecting entire applications. If this were the end of the story, then security in virtualized architectures would be limited to passive monitoring and the resulting security tools would remain crippled.

In this chapter, we introduce the Turret framework¹ that combines both types of active monitoring with passive monitoring to create a general secure active monitoring solution for security applications. We show how the monitoring mechanisms can be implemented and protected. Any system that uses active monitoring, including any future advances in the field, can benefit from the added protections that our work provides. The primary research contribution of this work is a framework to perform secure, active monitoring in a virtualized environment. We show design techniques that allow installation of protected hooks into an untrusted User VM. These hooks will trap execution in the untrusted User VM and transfer control to software in the Security VM. We also show design techniques to perform active monitoring from outside of the untrusted User VM. These techniques utilize hardware interrupt events such as network packets or user input from keyboards and pointer devices as active monitoring triggers. Finally, we show how active monitoring can leverage passive monitoring to create a generally applicable framework for security applications.

¹We call our framework *Turret* (pronounced Tûr'it) after the small towers that were used in medieval castles for military lookout and fortification.

Ensuring the security of this system is non-trivial. Our design is a departure from traditional secure systems work in that we place hooks inside the untrusted User VM, without layering them directly on top of trusted code. By limiting the functionality of the code placed inside the untrusted User VM and providing specialized protection mechanisms, we are able to ensure the security of this approach. The functionality removed from the untrusted User VM is then implemented in the Security VM, so the overall functionality of the system is not reduced. The protection mechanisms can be deployed as needed so that the system only uses more costly mechanisms when required. Using these techniques, we are able to thwart attempts by malware to disable security applications that use our monitoring framework. We provide a security analysis of our architecture in Section 3.6.

3.2 *Background Information*

3.2.1 Previous Approaches

Virtualization technology has made it possible to provide security services with better protection by isolating them into separate, protected VMs. Research on techniques like memory and disk introspection [69, 125] have shown how to leverage this isolation to securely monitor a system’s state. Virtual machine introspection (VMI) works by having a Security VM map the physical page frames of an untrusted VM into its own address space. It allows security applications to have complete visibility over another virtual machine’s raw memory state. Using VMI, higher-level code and data structures can be semantically reconstructed to provide an abstract view of the system’s state. While VMI has many applications, it is fundamentally limited because it can only perform passive monitoring. Therefore, introspection alone is not sufficient for applications that rely on active monitoring, such as anti-virus tools and host-based intrusion prevention systems.

Recent work on malware analysis [87, 117, 186] uses a form of VM-based active monitoring to intercept and analyze the run-time behavior of malware in a controlled environment. Although active monitoring is an integral goal of such systems, the requirements

and usage scenarios of these systems and our system differ, making malware analysis approaches unsuitable for use on production systems.

Malware analysis systems are primarily designed to monitor a large and comprehensive set of activities inside the User VM at a very low-level. This approach is feasible for two reasons. First, the offline fashion in which this analysis is normally done makes performance and run-time overhead a non-issue for such systems. Second, the fact that it is done in a staged, controlled environment, means that the collected low-level data can be directly mapped to the malware's activity with few false positives. In a production setting such as ours, the performance impact created by such systems make its use impractical. Furthermore, the low semantic level in which events are captured makes it difficult to infer the higher-level data needed to make security decisions.

In malware analysis, another important requirement is that the analyzer and the active monitor must remain hidden from the malware. This means that they should not introduce any noticeable side-effects in the malware's execution environment, as this could cause the malware to intentionally alter its behavior in an effort to thwart analysis attempts. However, in our production setting, we are only concerned with the protection and effectiveness of our monitoring components.

3.2.2 Xen Hypervisor

The Turret framework is designed to be sufficiently flexible to be compatible with a wide variety of execution environments. These include various virtualization platforms, co-processor platforms, and TPM platforms. However, the implementation of any such framework requires attention to low-level details, and as such it requires some effort to port it between platforms. For the purposes of our prototyping and testing, we choose to build Turret using the Xen virtualization platform [16]. In this section, we provide background information on the features of Xen that are most relevant to the Turret framework.

3.2.2.1 Xen Overview

Xen supports both the para-virtualized and the fully-virtualized methods of system virtualization. Para-virtualization consists of altering the guest OSes by replacing sensitive instructions that cannot be virtualized with special hypercalls, that is, calls that are made directly to the hypervisor. This approach has the advantage of providing good performance, since no trapping is done, and also allowing virtual machines to run on top of non-virtualizable architectures (such as x86) [138]. Nevertheless, one drawback of paravirtualization is that the guest OSes must be modified. For this reason, Xen also supports fully-virtualized systems which allows it to run unmodified OSes by using the Intel VT-x and AMD-V technologies.

Xen uses a split VM architecture, meaning that regular guest OSes are kept in unprivileged VMs (referred to as user domains or domU), and a single administrative VM exists (referred to as dom0). Dom0 can be seen as a VM level extension of Xen in which all of the management functionalities are located. It has complete access rights to all VMs being run and also works as a device driver proxy for domU's virtual devices.

The hypervisor itself is a simple and thin software layer whose main job is to guarantee proper isolation between VMs, performing minimal resource management. This isolation is quite robust, since Xen relies directly on hardware-level protection mechanisms and has a much narrower interface than a standard operation system (e.g., Linux), leaving less room for programming errors or bugs that could result in isolation violations.

3.2.2.2 Memory Management

One of the key tasks for a hypervisor is to partition the system memory between each VM. Xen achieves this using three levels of memory: machine, physical, and virtual addresses. The machine addresses are the actual addresses used by the hardware and managed by the hypervisor. The physical addresses are what each guest OS uses. This first abstraction allows the hypervisor to assign non-contiguous memory regions to a VM. As far as the

VM is concerned, the physical addresses are the addresses used by the hardware. In reality, these addresses are translated by a lookup table in the hypervisor into machine addresses. The third type of address is a virtual address, which is used the same way in the OS as traditionally used in OSes.

Both machine and physical addresses are often referred to in terms of a machine frame number (MFN) and a physical frame number (PFN). These numbers refer to a single page of memory, which are typically 4kB bytes each². A complete address is given as both an MFN or PFN and an offset into that page of memory. Using this scheme, Xen provides tables to convert from MFN to PFN and PFN to MFN. These tables are called M2P and P2M, respectively. Similarly, the running OSes use a page table (PT) to convert between virtual addresses and machine addresses. Xen protects these PTs in order to ensure the memory isolation between VMs. A para-virtualized OS must invoke a hypercall to modify its PT and a fully-virtualized OS will trap into the hypervisor when it attempts to change its PT.

3.2.2.3 *Inter-VM Isolation*

The hypervisor maintains distinct memory regions for each VM through PT management, as described above. However, there is still a need for some communication between VMs. For example, dom0 maintains the virtual hardware devices that are used by each domU. Therefore, every hardware event including keyboard events, mouse events, network packets, etc. are sent through dom0. Given the presence of this communication, one cannot argue that the isolation between VMs is complete. Instead, we refer to this isolation as being *controlled*. There are a limited number of ways to communicate between VMs so these communication channels can be carefully audited. Furthermore, Xen contains built-in mandatory access control (MAC) hooks that allow one to specify an appropriate inter-VM communication policy [146]. The narrowness of Xen's interface with each VM allows

²Here we refer to the x86 architecture where memory pages are usually 4kB, but can also be 2MB or 4MB.

for this policy to be much more simple than a typical OS-level MAC policy. As a result, we can have greater confidence that the policy and the interface are implemented correctly, and that the inter-VM isolation is sufficiently controlled to prevent an attacker from passing malicious code between VMs.

3.3 Security Requirements and Threat Model

We designed the Turret framework using a set of self-imposed design and security goals. This section introduces these goals in their idealized form. Later, as we describe Turret’s implementation details, we explain the compromises that had to be made to realize this framework. Understanding the idealized goals described below provides a foundation for understanding how the implementation compromises impact the overall security of the framework. We perform this security analysis in Section 3.6.

3.3.1 Design Goals

The Turret framework is based on six high-level design goals. In a general sense, these goals can be seen as typical good programming guidelines, or good security guidelines. For example, some of our goals could be seen as specialized versions of Saltzer and Schroeder’s classic security design principles [148]. This is intentional, as our goal was to leverage known design principles in order to build a robust monitoring framework. With this in mind, we identify the following six goals:

1. **No superfluous modifications to the hypervisor.** The hypervisor should remain as small and simple as possible since it is part of the TCB. If a hypervisor includes the necessary primitives to support the monitoring framework, then it should not be modified. If a hypervisor lacks the necessary primitives, then the modifications made should be what is minimally required to support the monitoring framework.
2. **No superfluous modifications to the VM or the user OS.** Modifications to the user OS (i.e., the OS being monitored), are problematic. Without careful attention to

security, the user OS and any malware running in it can tamper with this code. One of the key reasons why virtualization is attractive for secure, active monitoring is the controlled isolation between VMs. Placing monitoring code within the same OS that is being monitored bypasses this isolation. Therefore, this requirement encourages all monitoring code to remain in an isolated VM unless such a restriction makes it impossible for a monitor to gather the necessary information. In these situations, the monitoring code must be strongly protected to ensure that an attacker can not tamper with or circumvent the code.

3. **Small performance impact.** An excessive performance impact can render a monitoring framework worthless. This requirement ensures that the monitoring framework does not prevent the user OS from performing its intended functions. The performance impact is measured as any reduction in performance of an application caused by the monitoring software. Ideally this impact is both small and consistent, but some initialization costs may be required.
4. **Rapid development of new monitors.** New monitors may be needed to address new types of attacks. Furthermore, it is advantageous to keep the monitor code simple to limit the opportunity for introducing errors into the monitors. The monitoring framework should provide APIs that are used to develop new monitors. Therefore, satisfaction of this requirement means that the APIs should be designed in a way that simplifies the job of the monitor developer.
5. **Ability to monitor any data on user OS.** Monitors should have a full view into the user OS. The monitoring framework should not be limited to providing information about a small part of the user OS. For example, an ideal memory monitor should be able to view all memory on the user OS. Likewise, an ideal disk monitor should be able to view all data going to and from the disk device. While this ideal may not always be possible, the more information a monitor can view, the harder it is for an

attacker to evade detection.

6. **Target OS cannot tamper with monitors.** If the user OS can tamper with the monitors, then the possibility exists for malicious code to tamper with the monitors. For this reason, all of the monitors should be isolated or protected from the user OS. This is related to requirement (2), above. However, here we require that all monitor code, regardless of its location, be protected from attack. If all monitor code is in an isolated VM, then this is not difficult. If some monitor code must be placed outside of the TCB, then additional measures must be taken to protect that code. The extent of these measures will depend on the nature of the code being protected. In summary, the protection of the monitoring components should follow as closely as possible the formal requirements established in Section 3.3.2 for secure active monitoring.

We note that the last requirement does not enforce complete adherence with the formal requirements raised in Section 3.3.2 due to the difficulty of preventing all possible attacks, as will be discussed in Section 3.6.2. It does, however, significantly raise the bar and prevents the majority of attacks against active monitoring security tools.

3.3.2 Formal Requirements

In this section, we present a formal model that generalizes security applications performing active monitoring by placing hooks in a system to initiate actions when specific events occur. We use this model to analyze possible attacks on such applications under a powerful adversary that controls the entire system and identify a list of requirements that an ideal secure monitoring approach should satisfy in order to defeat such attacks. These formal requirements drive the design of the active monitoring hooks in the Turret framework.

Figure 1 illustrates our model. Consider a security application $A(C, D)$ with code C and data D that wants to actively monitor the occurrences of a set of events E occurring inside a machine M . Suppose that the application depends on libraries or OS subsystems denoted by $L(C', D')$ for its execution. In our model, we generically represent events as

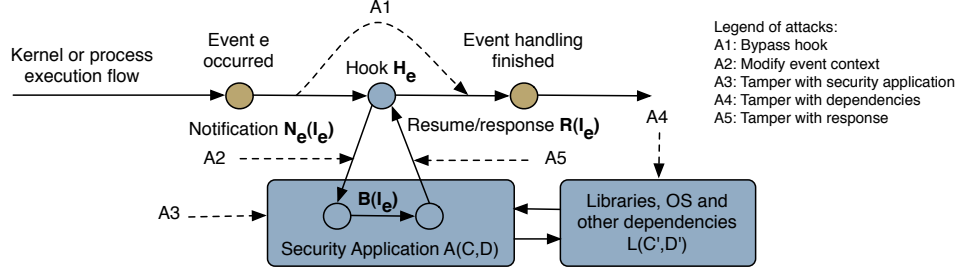


Figure 1: Formal model of secure active monitoring shown with potential attacks.

activities occurring sometime along the execution of the kernel or a user process, which are handled by event-handlers that exist in the system. Any event $e \in E$ is actively intercepted by placing a hook H_e in the control flow path between the point of the event occurring and the point where handling the event is finished. The purpose of the hook is to initiate a diversion of control-flow to the security application. Depending on where the security application resides, this diversion can be a straightforward control transfer, a process switch or even an inter-VM communication. Therefore, we use a generic notation N_e to represent the notification call to the security application. The context information I_e about the event and the hook is sent along with the notification. We express the behavior of the security application for the particular instance of the event as $B(I_e)$, which may include performing checks, processing models, generating logs, determining appropriate responses, etc. Finally, the response of the security application is denoted by $R(I_e)$, which are actions carried out on the system, including updates to the state of the system or modifications in execution flow.

We can identify several classes of attacks on various aspects of the active monitoring model. The first class of attacks (A1) disables or bypasses the hooks H_e or tampers with the notification mechanism, so that N_e is not invoked. Attacks (A2) can target and modify the context information I_e , providing the security application with an altered view of the occurred event e . In addition, some attacks may change the behavior $B(I_e)$ exhibited by application on receiving I_e . These include attacks that modify the security application A and its code C and data D (A3), or any of its dependencies L (A4). Attacks (A5) may alter

the response carried out by the security application by intercepting and modifying it.

The requirements for a secure active monitoring architecture that defeats the attacks are as follows:

1. N_e is triggered if and only if e occurs legitimately.
2. I_e is not modifiable between the occurrence of e and the invocation of N_e .
3. $B(I_e)$ of the security application is not maliciously alterable.
4. The effects of $R(I_e)$ on the system are enforced.

The first requirement states that an attacker should not succeed in circumventing hooks or generating spurious notifications. The second requirement ensures that an attacker cannot modify the context information I_e before invocation of N_e to alter or hide information regarding the event e . The third requirement ensures that the functionality of the security tool itself is not maliciously altered, defeating all attacks that tamper with the application process or any underlying subsystems it depends on. The fourth requirement ensures that the responses on the state of the system are always carried out as intended without letting the attacker modify them.

Assuming that an attacker has complete control over the system M , a security application that executes in the same machine with the same privileges as the attacker is unable to satisfy the above requirements. This is because the attacker can disable any protection mechanism and have complete access to the hooks, the application and its dependencies. By having higher privilege levels than the attacker, a hypervisor based approach can incorporate certain protections that the attacker cannot disable. Even in this scenario, if the security application is in the same VM as the attacker, it is hard to satisfy the third requirement. Since the application is running on subsystems controlled by the attacker, the run-time behavior of these subsystems along with the application needs to be protected, which may result in having to protect a large portion of the kernel. Although systems

like SecVisor [155] may be sufficient to protect the kernel code, they are not suitable for protecting kernel data. This motivates our architecture of having the security application execute in a separate security VM, which is isolated from the attacker.

3.3.3 Threat Model and Assumptions

We make the standard assumptions seen in most other virtualization security architectures [56, 68, 69, 91]. The hypervisor and Security VM are part of the trusted computing base (TCB) and the User VM is not. Therefore, malicious code can only affect the User VM. The hypervisor is ideally designed to be a small software layer that is both verifiable and secure. The hypervisor ensures isolation between the Security VM and the User VM, providing protection for security applications. Note that attacks such as Blue Pill [145] are not possible because a hypervisor using virtualization extensions is already installed as part of our architecture. Similarly, the SubVirt attack [100] is not possible because it was not designed to handle nested virtualization.

Beyond these general assumptions, we make some more specific assumptions for different portions of the framework. First we consider the runtime security of software placed in the User VM (e.g., software hooks). In order to focus on this problem, we assume that the machine can undergo a secure boot [9]. Furthermore, we assume that the User VM undergoes an *initialization* after boot. This initialization procedure will start the components, protect them, and provide for any additional security configuration. After the VM is initialized, it enters a *running* state where it is assumed to be subject to malicious software and other attack attempts.

Next we consider the active monitoring event handlers in the Security VM. In order to assure the integrity of our security software, we assume that it has a trusted execution environment (such as that provided by a secure virtualization environment or a platform implementing trusted computing [161]) that is isolated from malicious software running in the user environment. This trusted execution environment is created through the controlled

isolation provided by the hypervisor. This is why we consider both the Security VM and the hypervisor to be part of the TCB.

Finally, because our solution uses virtual machine introspection (VMI) to extract user-generated content and determine the meaning of hardware input events or hook events, we must assume that the layout of the operating system (OS) and application-level data structures we examine have not been altered. This assumption, which we call the *introspection assumption*, is common to most current VMI-based solutions [69, 91, 126]. It also represents a fairly high bar for the attacker because modifying the layout of these data structures would require updating all code in the system that uses them. Otherwise the affected OS or application would no longer function properly.

While the introspection assumption was necessary to complete the work presented in this thesis, it may not be suitable for all environments. For example, in high security situations, security applications may not want to assume anything about the layout of the data structures within an operating system or its applications. In these situations, it is not currently possible to extract rich semantic information about the running system (e.g., locations of widgets on the screen or the DOM tree from a web browser) because there is no way to verify the integrity of the system’s data or code. Christodorescu et al. [36] are working to address this problem by building semantic knowledge from the hardware, which serves as a root of trust. Approaches such as this increase the trust in a small amount of data obtained through VMI, but leave the majority of the information with an unknown level of trust. Hopefully this balance will continue to shift as more researchers address this problem.

In our threat model, we assume that the attacker can compromise the user’s OS and any application running inside it. Aside from the restrictions implied by the introspection assumption, he is free to execute arbitrary code, but cannot violate the security isolation provided by the trusted execution environment. We also assume that the attacker does not have physical access to the host, and hence cannot manipulate hardware events before they

reach the guest OS. Finally, we assume that the attacker cannot make modifications to the hardware (e.g., by flashing firmware, in order to generate fake hardware inputs). These assumptions reflect a realistic attacker who has successfully infected a user's system with malware and obtained administrative privileges.

3.4 The Turret Framework

Turret is a virtualization-based framework designed to facilitate the monitoring of an operating system and all of its applications and data at runtime. Turret provides active monitoring, which is needed to interpose and enforce security decisions on specific system events. Turret also ensures that the security-critical software components are protected from any form of tampering or circumvention.

3.4.1 Overview

Examples of applications that can benefit from Turret include anti-virus tools, anti-spyware tools, control flow-based intrusion detection systems, spam prevention tools, click fraud prevention tools, and nearly any other software tool that performs active or passive monitoring. The potential applications span beyond the realm of security tools and includes system administration, performance analysis, and software testing. All of these applications share a high-level operational flow that is supported by Turret. First, active monitoring produces some form of event notification (e.g., a hook in the user OS or a hardware event). Next, the application learns more about this event using some form of passive monitoring. Finally, some action is taken in response to the event (e.g., writing a log entry or stopping the event from proceeding).

The Turret framework is illustrated in Figure 2, with components of the TCB represented in gray. As shown in this figure, active monitoring can take several different forms. Applications that require fine-grained event notification can place hooks in the User VM that send hook events to the Security VM. Alternatively, or in addition, applications can monitor hardware events such as keyboard, mouse, or disk events; or applications can

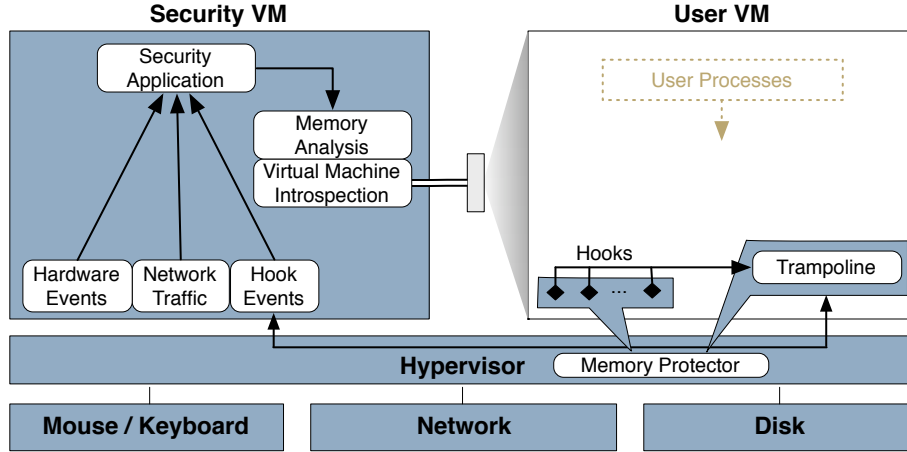


Figure 2: High-level view of the Turret framework and its core components.

monitor network traffic originating from or destined to the User VM.

There is a fundamental difficulty in conciliating protection and flexibility with active monitoring. The greatest flexibility comes when hooks are placed inside the untrusted User VM, which in a traditional scenario would make them prone to tampering by intruders with system-wide privileges. Solving this fundamental conflict is one of Turret’s contributions. At a high-level, it does this by splitting the security application into two VMs and using a special memory protection mechanism to guarantee the integrity of the hooks. As shown in Figure 2, Turret includes two VMs: the untrusted User VM and a Security VM that is part of our TCB.

Since the User VM is untrusted, software placed inside it requires special protection. This can be difficult to achieve if the components are too large or too integrated with the surrounding OS, so we keep them to the minimum required. These include the hooks for intercepting events, and a small specially-crafted trampoline code to pass events signaled by the hooks to the hypervisor. These components are self-contained and simple enough that write-protecting their memory footprint is sufficient to guarantee their correct behavior. We add a special mechanism to the hypervisor to provide these memory protections, along with an inter-VM communication functionality used for event passing. These additions, which we have implemented, are small to reduce the likelihood of introducing bugs into

the hypervisor.

Applications that don't need the added flexibility of placing hooks inside the User VM can opt to only use hardware and network events. In this case, the entire security application is contained within the Security VM, allowing the isolation provided by the hypervisor to improve the security posture of the security application.

The Security VM contains the core of the active monitoring application, where the processing and decision making associated with its functionality is done. Techniques such as virtual machine introspection can be used as part of this decision making to gather additional information about any of the event notifications. After a decision is made, the security application can perform any number of actions such as preventing the continuation of a function call in the User VM, writing a log message, or denying outbound network traffic.

As an example scenario, an anti-virus application would place its signature matching and containment algorithms in the Security VM, whereas its monitoring hooks would go into the guest VM. These hooks would be triggered whenever certain monitored events were executed by the User OS, and transmitted to the security VM by the trampoline with the aid of the hypervisor. The anti-virus' core engine would receive these events and use introspection to enrich them with contextual information, which would then be processed by its signature matching algorithms and heuristics. After reaching a decision, it would be sent back to the guest VM's trampoline, where a response measure is carried out, such as preventing a process from loading or a file from being written to disk.

The remainder of this Chapter will focus on the software components that enable secure, active monitoring in the Turret framework. This includes components in the Security VM, User VM, and the hypervisor. Chapter 4 will provide more details on the memory analysis techniques that enable useful passive memory monitoring through virtual machine introspection, as depicted in Figure 2.

3.4.2 Security VM Components

Most of the software necessary to support the Turret framework resides in the Security VM. The only software not in the Security VM is that responsible for installing and protecting fine-grained hooks in the User VM. Applications that do not require this fine-grained active monitoring can operate completely within the Security VM. The key benefit to placing software in the Security VM is that it is more difficult for malicious software in the User VM to attack it.

Each VM is largely isolated from other VMs through software-based isolation provided by the hypervisor. However, this isolation must not be complete because the Security VM requires access to the User VM's runtime state. Furthermore, the User VM requires some interface with the hypervisor in order to interact with the system's hardware. Different deployment scenarios (e.g., home user versus military installation) operate under different threat models. For the most secure settings, these interfaces must be minimized and audited to maintain the security of the Turret framework. For less secure settings, it may be sufficient to deploy Turret using an off-the-shelf virtualization solution. In either case, we refer to the isolation between VMs as *controlled isolation* to emphasize that it is not absolute and that it must be managed to ensure adequate security.

This controlled isolation makes it very challenging to deploy useful security tools in the Security VM. This section describes how the Turret framework overcomes the challenges related to accessing the information needed to support security applications, including both active and passive monitoring techniques.

3.4.2.1 Virtual Machine Introspection

Virtual machine introspection (VMI) is the technique of viewing information about one VM from inside another VM. In the case of the Turret framework, it is used to view information about the User VM from within the Security VM. This is strictly a passive monitoring technique, so it can be used either for a polling-based application (e.g., to periodically

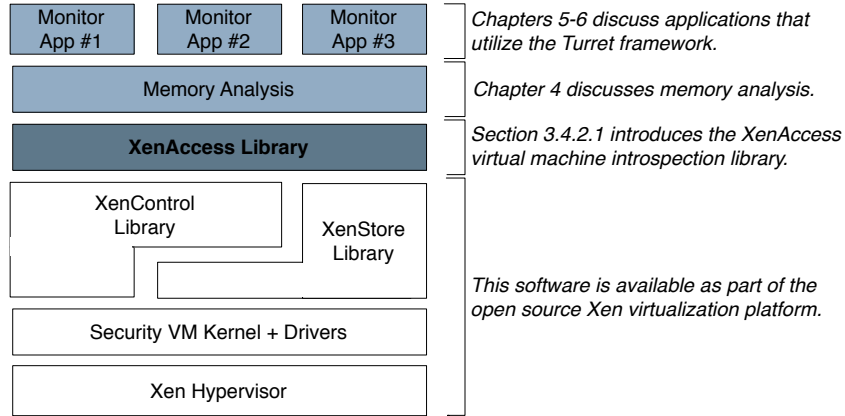


Figure 3: Turret’s virtual machine introspection capabilities are provided by the XenAccess Library. We originally created XenAccess to support this research, it is now available as an open source project at <http://www.xenaccess.org>.

scan memory for viruses) or to supplement event information for an active monitoring application.

In order to provide VMI capabilities to the Turret framework, we developed the XenAccess Library. As shown in Figure 3, the XenAccess library is designed to operate in the Security VM using the XenControl and XenStore libraries that are provided with Xen. Building on top of these libraries, XenAccess provides both read and write access to the runtime memory state of the User VM. Applications using XenAccess can access this memory using virtual addresses from the User VM’s kernel or from any of its applications (i.e., by specifying both the virtual address and the process identifier).

After requesting a specific virtual address, applications receive a pointer to the requested page of memory, mapped from the User VM, along with the offset into that page for the requested address. This low-level access leaves the application to interpret the data within each page of memory. A better solution, as depicted in Figure 3, is to place another layer of abstraction between XenAccess and the application. This memory analysis library can use domain-specific knowledge to extract useful information from the User VMs memory as described in Chapter 4.

3.4.2.2 *Hook Event Listener*

Hook events are the notifications that originate when the User VM triggers a hook installed in its kernel space. Section 3.4.3 provides details about how the hooks are installed and protected. In this section, we consider the complementary software that receives these notifications.

The Security VM contains the back-end components for processing hook events. This includes a kernel-level driver and a user-space security application that receives the hook events. The security application is where the decision-making functionality of the monitoring solution is implemented. It can be any software component that makes use of Turret, such as an anti-virus tool or a host-based IDS. The kernel-level driver is the communications agent responsible for relaying hook events between the User VM and the security application. These include hook notifications transmitted by the User VM and relayed by the hypervisor, and decisions sent by the security application to the User VM.

3.4.2.3 *User-Initiated Hardware Event Interposition*

As stated above, the Turret framework requires the ability to interpose on hardware events for one form of active monitoring. Virtualization has a similar requirement because these events must be multiplexed between the virtual machines running on a single computer. Building off of this invariant, the framework taps into the location where this multiplexing occurs. This ensures that applications using the framework have access to every hardware event. This mechanism amounts to a keystroke (and mouse event) logger running in the Security VM. However, unlike many keystroke loggers that are designed with malicious intent, we utilize knowledge of these events to improve the system's security.

It is important to emphasize that the hardware events received by the framework come *directly* from the hardware without passing through the User VM first. In most virtualization environments, this is handled in one of two ways. In Type I virtualization (where a

hypervisor runs directly on the hardware), the hardware interrupts go directly to the hypervisor and then they are either multiplexed from within the hypervisor or passed to a special virtual machine that multiplexes the events. In Type II virtualization (where a host operating system runs directly on the hardware), the host operating system receives the hardware interrupts and then multiplexes them to the virtual machines that are simply running as processes within the host operating system. Either way, the key point is that these hardware interrupts are received by the framework before being received by the User VM. This means that malicious software in the User VM will not be able to forge or modify any hardware events. The Turret framework builds on this property to ensure the overall security of the system.

3.4.2.4 Transparent Network Interposition

The final type of active monitoring supported by Turret is driven by network traffic. This traffic is redirected to a transparent proxy. The proxy can perform content analysis on the outgoing network traffic at the network layer or the application layer. It can use the receipt of a particular network packet as an active monitoring event. It can also determine if outgoing network traffic is authorized to leave the system. Similarly, Turret can analyze incoming network traffic; both for generating active monitoring events and for enforcing a security policy.

3.4.3 User VM Components

In traditional systems, all applications are run within a single operating system. The User VM fills the same role as this traditional operating system by running all applications that are not considered to be part of the TCB. The only exception is for the hooks and trampoline that are placed in the User VM to achieve some of the active control and monitoring capabilities provided by Turret. Any application can be run in the User VM since it runs a full featured operating system. With this in mind, the Turret framework can be used to protect a wide variety of systems including servers and desktop systems.

One of the key capabilities in the Turret framework is the ability to insert protected hooks throughout the operating system running in the User VM. These hooks can be jumps placed inside program code, redirections within jump tables, or any other technique that transfers control of execution. Hooks, or some other form of active monitoring, are required for any security software that stops malicious code prior to it doing any damage. This is because other techniques can only monitor by polling and are unable to guarantee detection at arbitrary locations in the code. With the protected hooks and other active monitoring techniques, there is a guarantee that the security software can evaluate an action before allowing it to happen. In the case of the protected hooks, this guarantee is provided by our memory protection mechanism, as described in Section 3.4.4.

When triggered, hooks redirect the system's control flow to another User VM kernel component, the trampoline. The trampoline is a specially-crafted piece of code that acts as a bridge between the hooks and the security driver running in the Security VM. It passes arguments from the hooked function to the hypervisor's inter-VM communication channel, which then delivers them to the security domain. The trampoline is also responsible for receiving commands from the Security VM to execute actions requested by the security software. As the rest of the User OS kernel is untrusted, the trampoline and the hooks must be protected from tampering. This need imposes several restrictions on the design and implementation of the trampoline. First, it must be completely self-contained. This means that its functionality must not rely on any kernel functions or global variables, since these may be compromised. It must also execute atomically at each round, in order to prevent scenarios in which race conditions are used to circumvent the monitor. Finally, the trampoline's usage of data elements must be completely non-persistent (i.e., not rely on data that was generated in previous hooks activations). As our protection mechanism currently does not support the protection of data regions, not following this requirement would make the usage of such data prone to tampering.

3.4.4 Hypervisor Components

Our framework requires two special features from the hypervisor: protection of User OS components and specialized inter-VM communications. The hypervisor modifications required to support these features are small, based on our implementation, which reduces the probability of introducing vulnerabilities into our TCB.

3.4.4.1 *Guest OS Component Protection*

The protection mechanism is one of the key pieces of Turret, as it guarantees the integrity of the user-space components of the framework. Unlike other framework components such as the VMI library and security application, which are isolated by the framework's inherent design, the hooks and the trampoline are situated in the User OS's untrusted kernel. Therefore these components require special protection to prevent an intruder from tampering with their behavior. This type of tampering could involve the omission or forgery of events, or disabling the monitoring solution. Because the trampoline is self-contained and the hooks are jumps or function pointers, marking each hook's memory as read-only is sufficient to prevent tampering.

A straightforward solution adopted by several OSes to write-protect memory regions is to simply guarantee that the corresponding entries in the page tables used by processes are marked with the appropriate permission – read-only, in this case. Although useful to prevent certain classes of failures, such as memory corruption bugs, this approach is not suitable for a security scenario. As we assume that the intruder can take control of the kernel, she can simply modify the page tables to disable these protections. Going a step further, if a protection agent is used by the OS to prevent direct modifications to page tables, the intruder could instead disable the agent to then alter the page tables. This argument can be inductively generalized to any number of protection stages and the bottom-line is simple: if the protection mechanism is based entirely in a single domain that is controlled by the attacker (e.g., the User OS kernel), it can be disabled.

In a virtualized architecture, the hypervisor is an ideal place to implement such protections for two reasons. First, as we assume it is part of our TCB, it cannot be tampered by a malicious user. Second, as part of its job in virtualizing the hardware, the hypervisor has complete mediation power over the memory mappings used by the VMs running on top of it. Our framework leverages this control to obtain a flexible, fine-grained memory protection mechanism. It is used to write-protect the hooks and the trampoline in the User OS's memory, so that no tampering can occur with these components. A graphical representation of this protection is shown in Figure 2. The strength of this protection derives from the strength of the TCB itself: the only way an attacker could undo it would be to compromise the hypervisor, which we assume cannot be done.

Our approach of protecting the hooks and the trampoline inside the User VM is generic. It does not rely on any features provided by the User OS. The complete mediation capability of the hypervisor over memory mappings of the User OS is sufficient. However, since the placement of the hooks may vary between different User OSes for the same events, the solution must take OS-specific architecture into account, and depending on the placement of hooks and the trampoline, different memory locations and data structures would need to be protected.

3.4.4.2 Inter-VM Communication

As our framework requires components located in different VMs to communicate, inter-VM communication functionality is needed. The trampoline code in the User VM must send the events it captures from the hooks to the Security VM, and the reverse path must also be traversed by replies sent from the Security VM. As virtualization inherently prevents VMs from directly interacting with each other, the implementation of such functionality must involve the hypervisor. The key property of the Turret communication mechanism that makes it different from existing generic mechanisms is that in Turret, the hypervisor must delay returning to the User VM until a response is available from the Security VM. A

benefit of this design is that the User OS will not be executing while we process the hook, which provides stronger guarantees for the system.

3.4.4.3 Related Work: Page-Level Memory Trapping

The Turret framework provides for fine-grained active monitoring using hooks placed inside the User VM. It also allows for active monitoring based on interposable hardware events in the Security VM. A related form of active monitoring allows the security application to set page-level trap events for User VM memory. This technique results in the security application being notified when the User VM accessing a given page a memory in a given way (e.g., accessing a page directory with write permissions). While not as fine-grained as placing hooks directly inside the User VM, this technique has the advantage of being easier to secure and easier to deploy across a wide variety of user operating systems. Page-level memory trapping has been demonstrated in the Ether malware analysis research project [50] and is available in VMSafe [173], a VMI library for VMWare products.

Page-level memory trapping could be easily integrated into the Turret framework. It would involve an additional change to the hypervisor, along with a supporting API within the Security VM. Once in place, it could serve as yet another active monitoring technique that could be leveraged by security applications.

3.5 Turret Implementation

The primary goal for the Turret framework is to satisfy the design goals and security requirements discussed in Section 3.3. We chose Xen as a virtualization solution because it is a Type I hypervisor; it runs directly on the hardware, allowing for a solid foundation to the TCB. It also already includes an infrastructure suitable for some of our monitoring needs, so that changes to the VMM can be kept to a minimum (design goal (1)). Likewise, by building on top of Xen's infrastructure, we only require changes to the User OS for fine-grained active monitoring hooks, allowing us to satisfy design goal (2). To prevent the

target OS from tampering with the monitors and satisfy design goal (6), we place the monitors in a different VM than the User OS. Active monitoring hooks are protected separately, as discussed in Section 3.5.5. Finally, we desire a framework that can monitor any data on the User OS in order to satisfy design goal (5). Turret currently provides passive monitoring capabilities for the entire memory space in the User VM, along with active monitoring capabilities for user input events, network events, and hooks placed directly in the User OS kernel. While this is not an exhaustive list of all data associated with the User VM, Chapter 6 demonstrates that this is a sufficiently useful set of data and events to build a wide variety of meaningful security applications. We examine Turret’s adherence to design goals (3) and (4) in Sections 3.7 and 3.8.

Figure 2 shows a high-level view of the Turret architecture. In this section, we will look at the implementation details of several key aspects of this architecture. First we look at the virtual machine introspection library. Next we discuss the framework mechanisms for receiving notification of hardware events and network traffic. Next we discuss the hooks and trampoline that can, optionally, be installed in the User VM to provide fine-grained active monitoring. Then we discuss the inter-VM communication mechanism required to support the hooks and trampoline. Finally, we discuss the memory protection mechanism required to protect the hooks and trampoline.

The Turret framework utilizes functionality included with Xen in order to reduce the implementation overhead and adhere to design goal (1). At this point it is important to emphasize that while we acknowledge that Turret’s functionality and its adherence to our design goals and security requirements are partially based on the infrastructure already provided by Xen, the Turret framework and the principles supporting it could be implemented on other virtualization platforms as well. The core functionality needed in the hypervisor includes mapping memory between virtual machines, protecting arbitrary memory regions in the User VM, and viewing VM-specific metadata (e.g., running kernel version). This functionality could be added to any modern virtualization environment, if it is not already

there, allowing for support of the Turret monitoring framework.

3.5.1 Virtual Machine Introspection

Virtual machine introspection involves accessing the memory of one VM from another. Xen provides a function in the XenControl Library that is used for this purpose and this functionality could be added to other hypervisors using a small amount of additional code. In Xen, the function `xc_map_foreign_range()`, maps the memory from one VM into another. After the memory is mapped, it can be treated as local memory, providing for fast monitoring capabilities. However, this function only operates using low-level machine frame numbers (MFN) to reference memory locations. In order to be useful to monitoring applications, this access needs to be done at the higher level abstraction of virtual addresses or even using symbolic information (e.g., exported kernel symbols). The primary goal of our virtual machine introspection library, which we call XenAccess³, is to provide this higher level abstraction in order to facilitate passive memory monitoring.

In order to convert a XenAccess API call into a call to `xc_map_foreign_range()`, XenAccess must perform several memory address translations. This requires additional information about the User OS which can be obtained from the XenStore, a database of information about each VM, and interpreted using some knowledge of the target operating system's implementation. The steps needed to convert a kernel symbol or virtual address into a memory mapped page are discussed below.

XenAccess is primarily implemented in C as a shared library with 3345 source lines of code (SLOC). XenAccess makes use of the XenControl Library (`libxc`) and the XenStore Library (`libxenstore`). The current version supports Xen versions 3.0.4 through 3.4.0 and works for both para-virtualized and fully-virtualized VMs.

³The source code for XenAccess is available – for anyone who may be interested in extending this work or validating our results – at <http://www.xenaccess.org>.

3.5.1.1 *XenAccess API*

XenAccess uses the `xc_map_foreign_range()` function to view the memory of another VM. Using this function eliminates the need to modify the hypervisor or the User OS, satisfying design goals (1) and (2). This function can be used to map a memory page from the User OS using its MFN. XenAccess uses this function for raw memory access and then builds up from there using address translation tables in the hypervisor and the User OS. For example, to convert a PFN to a MFN, XenAccess uses lookup tables that are provided by Xen. To convert a virtual address to a MFN, XenAccess uses the PTs in the User OS.

The XenAccess API contains a rich set of functions for accessing the User VM's memory. Each of these functions falls into one of the category of functions described below:

- **`xa_init_*`**(): The init functions initialize access to a specific VM given a VM name or identifier. It returns a pointer to an instance structure that is used for making subsequent calls to XenAccess API functions. All calls to `xa_init_*`() must eventually call `xa_destroy()`.
- **`xa_destroy()`**: Destroys an instance structure by freeing any associated memory and closing any open handles.
- **`xa_access_*`**(): The access functions map a memory page from the User VM into the local address space, and return a pointer to this memory. API users can specify which memory to map by specifying the machine address, physical address, kernel virtual address, kernel symbol, or user virtual address. This memory must be unmapped manually with `munmap()`.
- **`xa_read_*`**(): The read functions are convenience functions wrapped around the access functions. Each read function will return a 32-bit or a 64-bit value from memory in the User VM, given a particular memory address. These functions are useful for extracting short chunks of data (e.g., pointer values or counters) without the extra

code overhead associated with the access functions.

- **`xa_translate_kv2p()`**: This function performs a page table lookup of a kernel virtual address using the User VM PTs.

There are also some additional convenience functions for performing common tasks such as converting a kernel symbol to a virtual address or accessing the kernel data structure holding process information (i.e., `task_struct` in Linux or `EPROCESS` in Windows).

All users of the introspection library must begin with a call to one of the `xa_init_*`() functions. These functions initialize the `xa_instance` struct which holds information that is used throughout the introspection process. Any work that can be done “up front” and cached is held in this structure. This includes locating the address of the kernel page directory, initializing a handle to `libxc`, initializing a pointer to a PFN to MFN lookup table, determining if the domain is paravirtualized or fully virtualized, and more. Once a user is done with the library, a call should be made to `xa_destroy()` to free any memory associated with the `xa_instance` struct.

After initializing the `xa_instance` struct, one can use any of the other API functions listed above. Here we explain some representative functions in more detail. Starting with the simplest, the `xa_access_kernel_va()` function takes a kernel virtual address and returns a pointer to the memory page holding that address along with the offset to the specified address within the memory page. This address translation requires a PT lookup, which requires XenAccess to load three memory pages⁴. First, the page directory is loaded to find the location of the PT. Next, the PT is loaded to find the location of the address. Finally, the memory page holding the address is loaded and this page, along with an offset to the address, is returned to the user. Returning a shared memory page contributes to the good inter-VM memory copy performance shown in Section 3.7, which helps to satisfy

⁴A virtual address lookup may require loading a different number of memory pages depending on the address translation mode in use by the User VM. Three pages are required for the most common case of an x86 machine with PAE disabled.

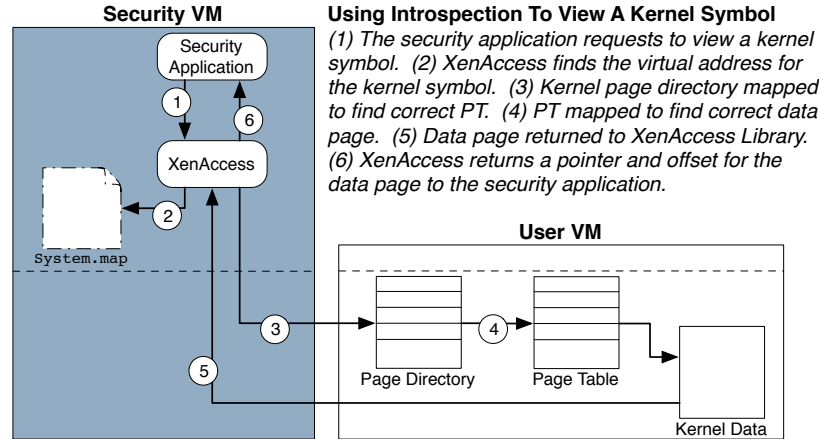


Figure 4: Steps needed to map a kernel memory page based on a kernel symbol using virtual memory introspection.

design goal (3).

The `xa_access_kernel_sym()` function, shown in Figure 4, requires one extra step beyond the virtual address translation described above. This step is to convert a kernel symbol to a virtual address. XenAccess performs this conversion using the exported kernel symbols associated with the kernel from the User VM. If the User VM is running Windows, the kernel export table is available in memory. If the User VM is running Linux, this information is extracted from the kernel's `System.map` file. Either way, XenAccess scans this kernel exports until it finds the symbol that the user requested. It then proceeds with a virtual address access using the address associated with the kernel symbol. Since under Linux this operation requires performing a lookup from a file on disk, it is considerably slower than the `xa_access_kernel_va()` function, but the results are cached so the average case is fast as discussed in Section 3.7. Further performance improvements could be achieved by memory mapping the file or moving the costly file read operations into the `xa_init_*` functions.

The final function that we will discuss is `xa_access_user_va()`. This function provides access to user space memory. Page table lookups for a virtual address in user space are essentially the same as kernel space. The main difference is that we must lookup the

location of the page directory associated with the process. Recall that for kernel space, the location of the page directory is cached during library initialization, but the page directory locations for each process can change as processes come and go. To lookup the page directory for a process, XenAccess scans the kernel task list looking for a process with the given process identifier. Upon finding a match, the page directory can be obtained from the `task_struct` or the `EPROCESS` structure in kernel memory, under Linux or Windows respectively. Using this page directory, the remainder of the virtual address translation is the same as previously described for the kernel.

3.5.1.2 Using Cache To Improve Performance

Since XenAccess must use memory from the User OS and the hypervisor to perform address translations, these operations can be costly. Therefore, XenAccess uses a least recently used (LRU) cache to store the results of the address translations. This is similar to a translation lookaside buffer (TLB). However, in the case of XenAccess, we also cache kernel symbol names since disk access is always a slow operation. This caching is critical to achieving acceptable performance and satisfying property (3), as discussed in Section 3.7. One risk associated with such a cache is that the information may become stale over time (e.g., a memory page can be swapped to disk, or a process terminated). For this reason, one idea for future work in XenAccess is to support cache flushing, either through a simple API call to allow the user to handle the cache or through an automated internal mechanism that flushes the cache if a value in the cache results in a failed memory page mapping.

3.5.1.3 Using OS-Specific Information

A kernel virtual memory address can be converted to a MFN without any knowledge of the OS in the User VM. This is because the address conversion is specific to the processor architecture and not to the OS. A PT lookup, which is required to perform this address conversion, starts by obtaining the address of the page directory. This information is stored in one of the control registers, CR3, of the User VM CPU context. Starting with the page

directory, one can complete a PT lookup and, therefore, find the MFN associated with any virtual address on a host. However, it can be difficult to determine what virtual address to access.

Identifying virtual addresses that are interesting requires some knowledge about the OS. One artifact of compiling a Linux kernel is the `System.map` file. This file is a listing of symbols exported from the kernel along with the virtual address of each symbol. Using this file, combined with the ability to access arbitrary virtual addresses, one can view and modify data such as the system call table, interrupt descriptor table, Linux kernel module (LKM) list, task list, and more. In Microsoft Windows, exported symbols are available in debugging libraries and in `ntdll.dll`. Of course, making use of these data structures requires knowledge of the data layout inside each structure. In Linux, this is determined by inspecting the source code and using technical references such as the kernel books by Bovet and Cesati [26] or Love [109]. In Windows, much of this information is available in technical references as well [143]. However, the broad field of locating, interpreting, and extracting useful information from memory – known as memory analysis – is still very young. We discuss this problem in greater detail in Chapter 4.

3.5.2 Hardware Event Interposition

In a default Xen installation, all hardware events pass through the administrative VM known as `dom0`, before being sent to a User VM. Within `dom0`, Xen creates an abstract model of the hardware devices to present each User VM with a common hardware interface. This model, known as the device model, is implemented in Xen using a modified version of Qemu [20]. Since the Turret framework uses `dom0` as the Security VM, we can simply tap into the hardware events as they pass through the device model.

We inserted code in two places in the Qemu device model, which is written in C, to

extract keyboard and mouse events. At each of these locations, the code first opens a connection to a FIFO server (described below). Once the connection is established, the keyboard or mouse information is serialized and sent to the FIFO server. The information sent includes the key or mouse button associated with the event, whether the button was pressed or released, and the screen coordinates associated with the event (only for mouse events). In addition, we perform a screen capture of the User VM using the `vga_hw_screen_dump` function in Qemu. The location of this screen capture file is sent along with the other event information. The screen capture is made available to applications using the framework and can be used for a variety of purposes including to assist in verifying user intent. After sending this information, Qemu waits for a reply before passing the event to the User VM.

These events are all received by a FIFO server that is part of the security application. The FIFO server is another process that is receiving these hardware events through an interprocess communication (IPC) channel. In this case, the IPC is performed through a UNIX named pipe. Under this design, the security application can setup a FIFO server to receive the events if it is interested in them. If it does not need the events, then it does not setup the server and the device model works as it normally does. If it does want to receive the events, then the device model connects to the server and sends the events.

Beyond keyboard and mouse events, the Turret framework can also use network events for active monitoring. The default setting in Xen is for all network traffic to pass through a virtual network bridge in dom0. Using standard network inspection tools, we can view all network traffic to and from the User VM. Each network packet can be treated as an active monitoring event. Or, alternatively, an intermediate proxy can reconstruct application-level semantics from the network stream and create events based on this higher-level abstraction (e.g., a transparent SMTP proxy can reconstruct an outgoing email message before sending the event notification to the security application).

3.5.3 Hooks and Trampoline

Detecting malware on today's systems requires monitoring events as they happen. This, in turn, requires placing hooks throughout the system being monitored. These hooks are usually numerous and placed throughout the kernel to detect operations such as process creation, writing to disk, network activity, and inter-process communication. The Turret framework is capable of placing hooks anywhere within the kernel of the guest OS. Hooking standard system calls requires the memory protections described in Section 3.5.5. Placing hooks in other locations requires additional protections as described in Section 3.6. Regardless of the hook location and its protections, the implementation of the hook processing system is the same.

The hooking techniques for Turret can be used in any operating system. However, our implementation focuses on Windows as this is the most commonly used user operating system. In order to describe the hooking process in a concrete manner, we describe how it works for one particular system call in Windows. The `NtCreateSection` system call is used to create a new process and is therefore interesting from a security viewpoint. The mechanism required to hook this system call is the same as hooking any other system call and very similar to hooking an arbitrary location within the kernel. Furthermore, the techniques used to process this hook are similar to what one would use for processing any hook.

In order to install the hook into the kernel API `NtCreateSection`, we implemented a Windows kernel driver called `hookdriver.sys`. Upon installation, which happens during the User OS initialization, the driver creates a trampoline code section, modifies the appropriate system service descriptor table (SSDT) entry to point to it, and informs the hypervisor to activate necessary memory protections for the hook. The driver's implementation has 324 source lines of code (SLOC), and the trampoline occupies 89 bytes of memory.

The trampoline code is placed in a page of memory allocated from the nonpaged memory pool by calling the kernel function `ExAllocatePoolWithTag`. This ensures that the trampoline is always available, and will never be swapped to disk. The trampoline code section is copied from within the driver's code base to the newly allocated memory region. In order to modify the SSDT, we first identify the index of the `NtCreateSection` service. Then we identify the base of the SSDT using the kernel symbol `KeServiceDescriptorTable`. We then create the hook by placing the address of the trampoline in the appropriate entry after storing the old service routine's address (i.e., the location of the actual `NtCreateSection` function) in a pointer. This pointer is placed in the newly allocated memory region along with the trampoline code, so that it is protected from malicious modifications.

Once the hook is placed to point to the trampoline code, the driver initiates a notification call using a `VMCALL` to the hypervisor to inform the installation of the hook and the address range of the newly allocated memory region. This information is used to secure the indicated regions using the `prot_range` hypercall described in Section 3.5.5. This entire process is completed during the secure initialization of the User OS.

3.5.4 Inter-VM Communication

When the trampoline code from the User OS makes a `VMCALL` into Xen, it is sending a signal asking Xen to assist with inter-VM communication. In the Turret framework, inter-VM communication is facilitated by Xen with signaling from the VMs performed through hypercalls. We added a new hypercall to Xen, `turret_op`, that is callable from both the User VM (via a `VMCALL` instruction) and the Security VM (via a direct hypercall). This hypercall takes two arguments. The first argument is a command. If the command requires a parameter, it is sent as the second argument. In the Security VM, the security application issues its hypercalls using a *security driver*. The security driver, described in more detail below, is a generic hypercall pass-through provided with the Turret framework that will work with any security application. We provide details on each command below.

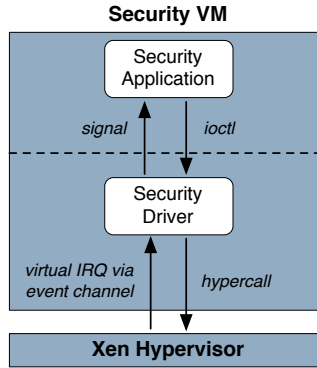


Figure 5: The information flow path from Xen, through the security driver, to the security application and back are all event driven to provide good performance when processing hook events from the User VM.

The `TURRETOP_security_register` command saves a memory address of the buffer used to exchange information between Xen and the security driver. The other two commands are slightly more complex.

The `TURRETOP_guest_hook` command builds a `struct` to send as a request to the security driver. This `struct` contains a unique identifier for the request and information about the hook event (e.g., hook number, associated Windows handle, or process id). This `struct` is copied to the security driver's shared memory region and then a virtual interrupt is sent to the Security VM. This virtual interrupt, which is implemented using Xen event channels, is a signal to the security driver to process the hook information in its shared memory region. At this point, the command waits at a barrier until a reply is provided by the security driver. After the reply is provided, it is returned causing the `VMCALL` instruction to return, which allows the User VM to continue normal operation.

The reply from the security driver is signaled with the `TURRETOP_security_response` command. Upon receiving this command, Xen gets the reply value by copying a `struct` from the security driver's shared memory region. Next, the command makes this reply available to the hook command and breaks its barrier.

These three commands, implemented as a single hypercall, are all that is needed to support the inter-VM communication for the Turret framework. They were implemented

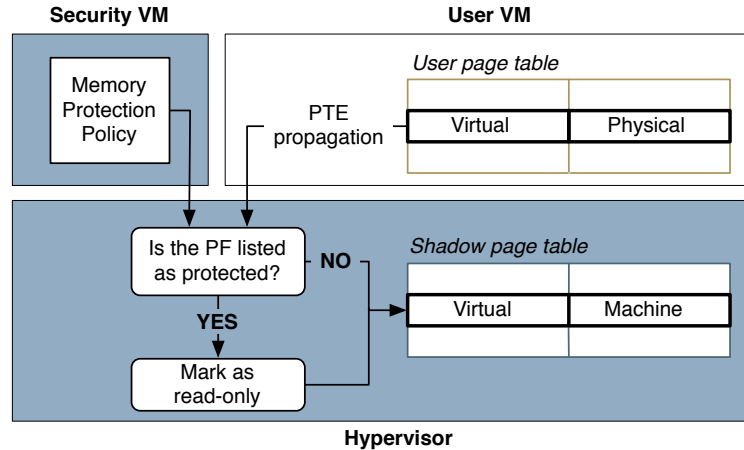


Figure 6: The page protection scheme leverages the propagation of page table entries (PTEs) from guest space to hypervisor space to protect memory pages.

by adding 127 SLOC to Xen.

The security driver is designed to pass information up from Xen to the security application, and down from the security application to Xen as shown in Figure 5. No information processing or decision making occurs within this driver. This is an intentional design choice because it is harder to implement and change kernel-level code. As new features are added to this system, changes will typically only be made to the security application.

Since the security driver is designed to run in the Linux kernel, it is implemented as a Linux kernel module (LKM). The LKM is installed automatically when the security application is started. During initialization, the LKM sets up a shared memory region, a `proc` entry and its handler, and a virtual interrupt handler. The shared memory region is used to pass data between the security driver and Xen, as described above. The `proc` entry receives data from the security application and the virtual interrupt handler receives signals from Xen. The security driver is 182 SLOC and should not require any changes when new hooks are added to the system.

3.5.5 Memory Protection

We leveraged Xen’s memory management subsystem when building the memory protection mechanism. The primary goal of memory management in Xen is to virtualize each

guest OS's view of memory and enforce isolation between the OSes. In fully-virtualized VMs, Xen does this using a technique called *shadow paging*. This technique maintains two versions of page tables for each VM: guest page tables (GPTs), which are controlled by the guest; and shadow page tables (SPTs), which are controlled by the hypervisor. The guest OS handles its GPTs the same as it would in a non-virtualized setting. The main difference is that the GPT's mappings translate virtual addresses to an intermediate layer of addresses, called *physical addresses*. Physical addresses virtualize the memory view of a guest OS, similar to the way virtual addresses work for processes. SPTs provide direct mappings from virtual to machine addresses, which are the addresses used by the hardware. Therefore, the SPTs are used by the hardware to translate addresses for the guest OS while Xen maintains consistency between the GPT and SPT. When an entry is added or changed in a GPT, Xen translates the physical address into its corresponding machine address, performs any necessary adjustments, and then updates the corresponding SPT. This process is called page table entry (PTE) propagation. Under this model, Xen controls the actual machine frames used by each VM, while also providing each guest OS with the illusion that it has full control of the memory.

Our memory protection mechanism protects arbitrary memory regions with a byte-sized granularity. It is composed of two main parts. The first is implemented in the `_sh_proagate` function, which controls the propagation of entries from GPTs to SPTs. The second is implemented in the `sh_page_fault` function, Xen's page fault handler. Our mechanism adds 78 SLOC to Xen, satisfying our design goal of making only minimal additions to it.

The first part, illustrated in Figure 6, implements the core technique behind our protection mechanism. At this location we intercept the propagation of entries between the GPTs and SPTs, and then write-protect designated frames of the User OS's physical memory. This can happen whenever an entry is modified in a GPT, either legitimately or by an attacker. Since Xen has full mediation over propagation and is isolated from the the

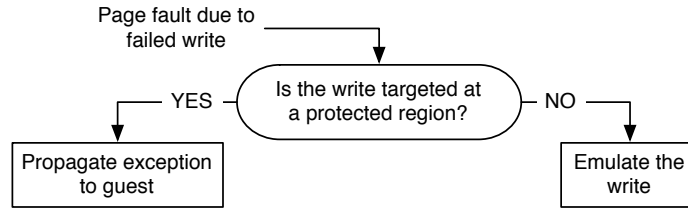


Figure 7: The page protection scheme leverages Xen’s page fault handler to make sure it can be done with a byte-sized granularity by emulating the memory write.

User OS, such protection cannot be circumvented. We store a list of the memory regions that require protection, which we call the protection list. Each time an entry is propagated from a GPT to an SPT, this list is searched for the entry’s physical frame. If it is found, its corresponding shadow copy is marked as read-only. This prevents the User OS from performing any further modifications to this page frame.

By itself, this technique only provides page-level protection, which is problematic if a page contains protected and writable regions. The second part of our mechanism extends this technique to provide byte-level protection. Its operation is illustrated in Figure 7. Each time a page fault occurs due to a failed write, we check the target’s virtual address, which is stored in the `cr2` CPU register. Next, we check the protection list to see if the target address requires protection. If it does, a page fault exception is propagated to the User OS, preventing the write attempt. If not, then the User OS is attempting to write to a non-protected region of a frame that contains a protected region. In this case, we emulate the write operation for the User OS.

We added a new hypercall, `prot_range`, that can be called from the security application running in the Security VM to initialize the protection list. This is done during the architecture’s initialization, as soon as the hook and the trampoline are placed inside the User OS. Additional memory ranges can also be added to the list at run-time, if desired. Each time an update is made to the list, the shadow page cache is erased to eliminate outdated mappings and force the re-propagation of new mappings. Since most applications, including our prototype, only add items to this list during initialization, there is no runtime

performance impact on the system.

This combination of page-sized memory protection and write emulation allows us to efficiently implement the protection of arbitrary memory regions of the User OS, with the granularity of a single byte. In our prototype, we used this mechanism to protect several memory regions in the User OS. The first was the `NtCreateSection` hook placed in the SSDT, a 4-byte long function pointer. The second was the trampoline, a segment of code consisting of 89 bytes in a memory page allocated when the architecture is initialized. Additional components that require protection to prevent hook circumvention are discussed in Section 3.6.

3.6 Security Evaluation and Analysis

New security frameworks should, ideally, be secure against current and future attacks. In this section we evaluate how the various pieces of the Turret framework work together to ensure the security of the application in the Security VM along with all of the components required for its proper operation, including the hooks and trampoline code in the User VM. We start with a practical look at how existing attacks can impact Turret. Then we perform a more abstract analysis of Turret’s security against an unrestricted adversary.

3.6.1 Current Attacks

Several components of the Turret framework are sufficiently new that there are no current attacks that will effect them. In particular, all of the software components in the Security VM are protected by the isolation provided by the Xen hypervisor. Altering or disabling these components would require exploiting a software vulnerability in Xen. Only a few such vulnerabilities have been discovered, and these are always a result of software bugs that can be easily patched. Furthermore, this type of attack is outside of the scope of our threat model as discussed in Section 3.3.3.

Within the User VM, malicious code could try to hide or obfuscate data in memory from the `XenAccess` virtual machine introspection library. While no current attacks are

known to do this, the attack seems technically plausible. We note that such attacks are also outside of the scope of our threat model, but we will discuss them briefly in Section 3.6.2.

The attacks within the scope of our threat model include anything that an attacker can do from the User VM. Within this context, we are primarily concerned with attacks against the hooks and trampoline code. There are current instances of malware that perform context flow attacks to circumvent hooks placed within the OS kernel. Below we consider how this style of attack would impact the Turret framework.

3.6.1.1 Attacking User VM Hooks

An essential component of a successful attack is to evade detection by either hiding itself or by disabling defensive measures altogether. Many malware today incorporate features, similar to Agobot [168], to disable commercial anti-virus programs. Earlier methods involved process termination system calls for known anti-virus process names. However, most anti-virus programs incorporate hooks into process termination and creation routines to monitor their usages patterns for self-defense [165]. In addition, hooks are placed into events such as file/disk access or Windows registry updates for detecting malicious updates to the system. To defeat such systems, malware programs incorporate various rootkit methods [83] that can remove such hooks to successfully disable anti-virus tools prior to infection. A recent work [185] allows automatic analysis of the hooking behavior of malware by identifying the modified code and data structures. A large number of malware place their own hooks in order to hide their processes, drivers, files (e.g. the FU rootkit, NT rootkit etc.) and their malicious changes in the system or backdoors (e.g. Uay Backdoor). A classification of rootkits can be found in [144].

As described above, a technique used by attackers to defeat commercial anti-virus tools is to replace the hooked function pointer in the SSDT to either the original function pointer

(disabling the hook) or to a malicious function pointer that in turn calls the original one (hijacking the hook). In order to perform this attack, the malware must have detailed knowledge of the internal workings of these tools. Since these kinds of low-level attacks must be specially crafted for the target security tool to be effective, we were not able to test our memory protections with pre-existing attack code. Instead, we developed a synthetic attack that performs the hook hijacking attack using the same technique used by rootkits.

In our implementation, the initialization procedure in the User VM installs a hook in the SSDT. This hook is effectively changing a function pointer so that calls to `NtCreateSection` are redirected to our trampoline code. This hooking procedure is identical to that used by commercial security tools (e.g., anti-virus products) on the market today.

Our test system consisted of the User VM running with the trampoline and hooks initialized. To ensure the synthetic attack works properly, we then ran it without any memory protections enabled. During this test, the synthetic attack worked as expected, hijacking the hook and effectively preventing any execution of the trampoline code. Next, we repeated the test with the memory protections enabled. This time the synthetic attack failed to complete its installation because it was unable to change the write-protected entry in the SSDT and the trampoline code continued to execute normally.

This test shows that the memory protections work properly. Similar attacks could be constructed to modify the trampoline code, the pointer to the SSDT in the system service dispatcher, or the pointer to the system service dispatcher from the IDT. However, these attacks would also fail because these regions of memory are also protected by our system.

3.6.2 Future Attacks

In this section we consider how malicious software could attack the Turret framework once attackers have an understanding of how the framework operates. We start by briefly considering some attacks that are outside of the scope of our threat model in order to understand

how challenging it would be to perform these attacks. Then we compare framework’s implementation of the hooks and trampoline code in the User VM to the formal requirements presented in Section 3.3.2 to understand how well this code in the User VM is protected against future attacks.

3.6.2.1 *Attacking Virtual Machine Introspection*

While our threat model makes an *introspection assumption* stating that the OS and application-level data structures in the User VM memory have not been altered, it is still instructive to consider what it would take to perform this type of attack. The most security critical data is located in the User OS kernel, and this is also the most difficult data to alter. Altering the data can take one of three forms. The first is to simply change the data in place. This would result in the security application getting faulty information, but it would also impact the operation of the User OS that relies on this data. A second option would be to modify the format of the data structure. This would invalidate the security application’s knowledge of the data layout and prevent it from accessing the data it needs. However, doing so would also require modifying all of the code in the running User OS that accesses this data to understand the new format. Modifying the kernel code on-the-fly is challenging and risks detection. Finally, the data could be moved to a new location in memory (perhaps in addition to altering it’s format). Again, this would disrupt the security application, but also require changes to the running User OS. Given the challenges of modifying kernel data to hinder the use of VMI, we believe that this type of attack is unlikely.

Application-level code can be attacked in all of the same ways as the kernel, however here the task is somewhat easier on the attacker if the application can be restarted without arousing suspicion. This can often be done by simply crashing the application and forcing the user to restart. Given the current state of software reliability, most users will not suspect malicious activity in this scenario. By restarting the application, an attacker can more easily alter the data layout of that application in memory (e.g., by replacing the application’s

binary on disk before the user restarts it). However, in this case there are alternative security techniques that could detect such alterations and prevent the application from starting.

While there have been some concerns in the literature over the use of VMI for security oriented applications. Our analysis here shows that the introspection assumption appears to be valid for typical user scenarios.

3.6.2.2 Generating Malicious Hardware Events

Our threat model also indicates that an attacker can not access, or modify the hardware (including the firmware associated with the system's hardware). However, if an attacker could modify the hardware, then it is plausible for an attacker to generate or suppress hardware events. If an attacker can interfere with the specific events that are used by the security application to perform active monitoring, then the attacker could avoid detection. However, that attacker would need to ensure that critical hardware events were still delivered to the User VM or the malicious software would quickly be detected (e.g., if the keyboard appeared to stop working because malware overwrote the keyboard firmware to suppress the delivery of key stroke events).

A more plausible scenario would be for an attacker to specially craft network traffic to exploit a vulnerability in the security application. For example, if the security application was parsing network packets and the parsing code contained a vulnerability then an attacker could potentially exploit that vulnerability. This style of attack is very realistic however it has limited scope and impact. The software vulnerability could be patched and the attack would no longer work. In general, software vulnerabilities such as this can happen but they do not necessarily represent a flaw in the framework design.

3.6.2.3 Attacking User VM Hooks

In the Turret framework, all of the software in the Security VM are inherently protected from any type of disabling or tampering (attacks A3 and A4 in Figure 1) initiated from the intruder in the User VM. This can be generalized to our framework as a whole, as any

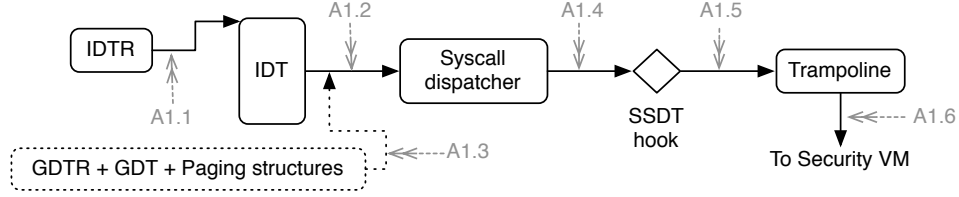


Figure 8: Various forms of attacks aimed at circumventing the fine-grained active monitoring hooks installed by Turret in the User VM.

application that makes use of Turret will be in the Security VM, and therefore integrated with the TCB. As such, the only option for an attacker would be to target the guest-space infrastructure (attacks A1, A2 and A5 in Figure 1) on which the security application relies on for active monitoring.

We first discuss various forms of A1 attacks illustrated in Figure 8, which are aimed at circumventing our monitoring infrastructure. They work by maliciously modifying the User OS components and the system structures that they depend on. In case of the SSDT hook and the trampoline, disabling or tampering with such components (attacks A1.5 and A1.6) would mean maliciously altering their memory state. For example, the hook could be replaced by another that points to a malicious function, or the trampoline could have critical parts of its code (such as the `VMCALL`) erased. Attacks of this type against the hook and the trampoline are effectively blocked by our architecture, as we write-protect their memory region. This fact has been empirically verified in the experiments we conducted, presented in Section 3.6.1. We can also generalize this property to any type and number of hooks in the kernel, as our hypervisor-based memory protection is done at the byte level and can be applied anywhere in kernel code or data.

Other types of circumvention attacks illustrated in Figure 8 involve manipulating certain kernel structures which control the kernel’s flow of execution between the moment at which the `NtCreateSection` event happens and the hook is triggered. In the case of our SSDT based hook, the IDTR register (A1.1), the system’s IDT (A1.2), the system call dispatcher (A1.4) and current address translation structures (A1.3) can be targeted. The latter includes

the GDTR register, the GDT and the current page table. These registers and structures are defined by the x86 architecture. The first three determine the flow of execution immediately after a system call is triggered by the `int CPU` instruction. In our prototype, we write-protect the IDT and the system call dispatcher code in memory, nullifying attacks A1.2 and A1.4. Such protections have no negative side-effects, as these structures are normally not supposed to be modified at run-time. For the IDTR, we were unable to implement a similar kind of protection, as changes to this register cannot be trapped by a hypervisor using Intel VT-x. Instead, we check its contents for modifications at every exit from the guest to the hypervisor. These exits (called VM exits) happen at least at every context switch, making attacks targeted at this register extremely difficult to succeed due to the short window of opportunity. However, for the AMD SVM platform, this attack is not a concern because changes to the IDTR are trapped by the hypervisor.

A more sophisticated type of circumvention can be done by manipulating the GDTR, the GDT and the system's page tables, as these are used in the translation step between the IDT and the system call dispatcher (attack A1.3 in Figure 8). An attacker could tamper with such structures to manipulate the address translation, and redirect the flow of execution to a different physical address, containing malicious code. Memory introspection is affected by a similar issue, as it normally uses page tables inside the User OS to perform memory translation. Although possible, such an attack would nevertheless be considerably difficult to implement in this particular case, as the dispatcher shares a single 4MB page with the rest of the Windows kernel. To succeed, an attacker would thus need to relocate a large critical portion of the kernel without detection—a considerable, if not impossible effort. In situations where such an obstacle is absent, a general solution to this problem would involve monitoring individual shadow page table entries to ensure that they always point to specific, known good locations.

In addition, a notification should be sent by the trampoline only if an event happens, that is, an intruder should not be able to send bogus notifications. This could be done, for

instance, by explicitly invoking the trampoline or jumping into arbitrary locations inside it (for instance, the VMCALL). Although this condition is not addressed by our prototype, we recognize it as a significant issue. One possible solution would be to first control the origin of branches by marking the memory region where the trampoline code resides with the non-execute (NX) bit. By doing so, every access to the trampoline would generate a fault, allowing us to check if the EIP value (the location from which the call was made) corresponds to an authorized hook location. If not, then we would know that the attacker is trying to make a bogus call to the trampoline and block it. The cr2 register could also be monitored, which would allow us to check the destination of the branch, and enforce a single entry point into the trampoline code. This would in turn block any attempts of branching into arbitrary locations of the trampoline code.

Other types of attacks can be avoided by disabling interrupts system-wide during the execution of the trampoline. This ensures that no other kernel thread is executed before the VMCALL. This guarantee, combined with our assumption that this system is running on a single processor, ensures that no one can change the monitored thread context to bypass the hook. Therefore, the code execution from the occurrence of the event up to the notification sent by the trampoline cannot be preempted.

This technique also automatically prevents A2 attacks, as interrupt disabling prevents an attacker from modifying any context information from the moment the event happens to the moment a response is received by the trampoline. Attack A5 is also prevented since the code responsible for carrying it out is already protected in the trampoline, and the fact that interrupts are disabled guarantees that its execution cannot be preempted. Although the triggering of non-maskable interrupts (NMIs) could circumvent the interrupt disabling and break the desired execution atomicity, we expect these to occur only during hardware fatal errors. By assuming the use of a single CPU core per VM in our prototype, we avoid the scenario in which an attacker could use a second core to explicitly send an NMI to the one running trampoline code and interrupt the execution flow. This assumption also prevents

the occurrence of other race conditions involving multi-core architectures.

We acknowledge that some of the anti-circumvention techniques mentioned above are very specific to the type of hooking implemented. In particular, the kernel code and data structures that link the actual system call event to the SSDT hook itself, are relatively few and easy to protect, enabling us to create a protected chain. But in a more general scenario, where hooks can be placed in code or arbitrary data structures, creating an equivalent protection chain can be more complicated. By patching kernel code whose execution precedes the execution of a code hook, for instance, an attacker could jump around it. Existing solutions, such as SecVisor [155], could be integrated with our architecture to guarantee the kernel’s code integrity and avoid this type of circumvention. Data hooks in arbitrary kernel data structures present a more interesting challenge because of data’s volatile nature. But existing approaches like passive monitoring of kernel control data structures [131], mediation of changes to kernel data structures [183] and semantic integrity checking [130] could be used to raise the bar for an attacker. Although these techniques certainly help mitigate the more generic circumvention problem, their kernel-pervasive nature would add a significant performance impact to the overall architecture. In this case, a compromise between security and performance exists, which permits each application to make an appropriate tradeoff given its needs.

3.7 Performance Evaluation

This section focuses on select micro-benchmarks that highlight the performance characteristics of the Turret framework. We discuss performance results for both passive monitoring and active monitoring. We discuss macro benchmarks and other application-specific performance tests later, in Chapter 6. Our tests show that the performance impact imposed by the Turret framework is very reasonable, adding only small latencies to the system performance and user interactions. We feel that these latencies are acceptable given the security benefits afforded by the Turret architecture.

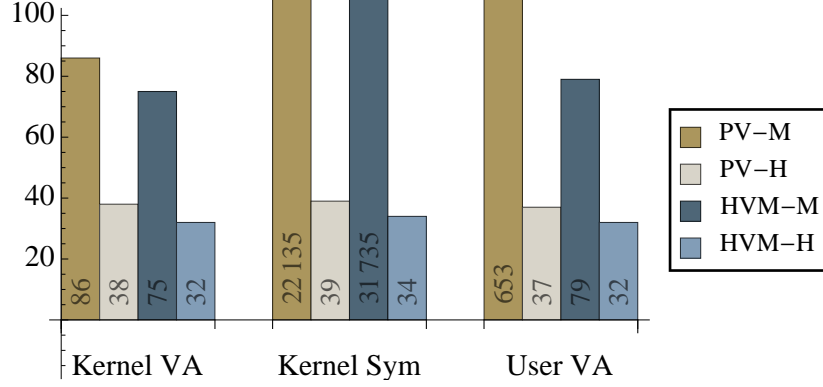


Figure 9: Performance of three memory access functions in XenAccess for a paravirtualized (PV) and hardware virtualized (HVM) VMs. Both cache misses (-M) and cache hits (-H) are shown. All times are in μ secs.

We performed the testing on a system with a dual core 2.33 GHz processor with 2MB L2 cache per core (Intel Core Duo T2700). The system had 2 GB of RAM, and an 80 GB 7200 RPM hard disk drive. The Security VM (dom0) was assigned 2 processor cores, and the User VM was assigned one processor core.

3.7.1 Passive Monitoring

Each performance measurement in this section was performed using the `gettimeofday()` function, which has a micro-second granularity. Times were measured by recording the time immediately before and after the function being measured. The difference between the two times was recorded. This measurement was repeated for 1000 times for each test. We choose 1000 measurements because this was sufficient to minimize the standard deviation for a given set of measurements under this setup. Additional measurements did not improve the precision.

The data in Figure 9 show the average time to complete the specified function call. The cache hit columns represent the results with the LRU cache enabled. The cache miss columns represent the results with LRU cache disabled. The simplest case is shown on the left of this graph. The `xa_access_kernel_va()` function must map three memory pages on a cache miss and one on a cache hit. This difference explains the improvement

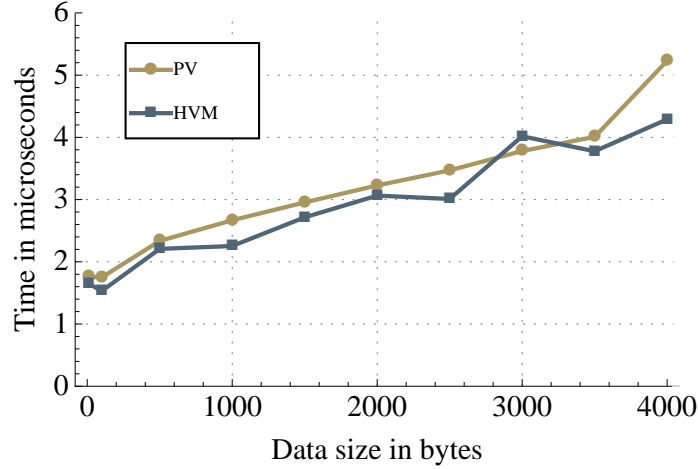


Figure 10: Time for monitor to read memory through introspection.

seen with the LRU cache. The time for `xa_access_kernel_sym()` is dominated by the operation to lookup the kernel symbol. This operation is a lookup inside a file on disk, which is costly. With a cache hit, the symbol to machine address mapping is stored in the cache, making the performance similar to `xa_access_kernel_va()`. The last access function is `xa_access_user_va()`. This function must traverse the task list in the domU kernel to locate the page directory for the process virtual address. This explains the slower performance for the cache miss. On a cache hit, this traversal is not needed, performance is essentially the same as `xa_access_kernel_va()`.

After the memory is accessed, the next step is to read from or write to that memory. As seen in Figure 10, this operation is fast compared to mapping the memory. These performance results show the time required to `memcpy()` data from kernel memory in the target OS. In general, we found that data is copied into a data monitor at a rate of approximately $1\text{kB} / \mu\text{sec}$. Figure 10 shows that `memcpy()` performance for PV and HVM VMs is essentially the same. The variance in these measurements can be attributed to experimental noise given the precision of our timing mechanism and the small measurement times. Looking at the cache hit values in Figure 9 and the memory copy performance, the memory introspection capabilities in XenAccess perform well enough to have a negligible impact on the

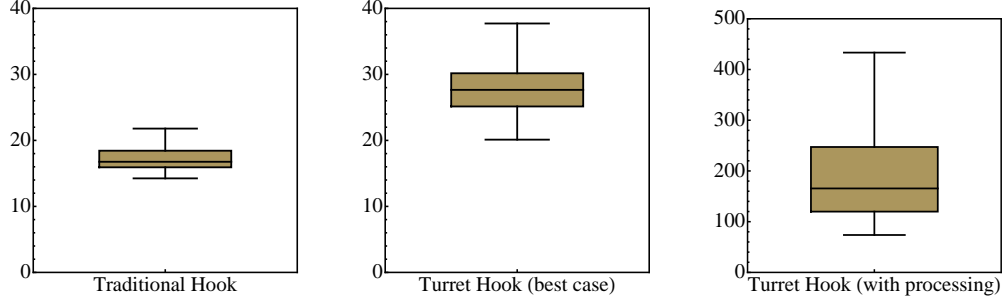


Figure 11: Hook performance is shown in the three charts. The traditional hook shows the time for processing a hook locally. The Turret (best case) shows the time for just sending a notification to the Security VM. The Turret (with processing) shows the time for processing the hook in our prototype application, using introspection to lookup file names from handles. All times are in μsecs .

overall system performance.

The performance results for VMI provide some insight into the types of applications that are best suited for introspection. The best applications map a single or small number of memory locations and view them over time. These types of applications only perform the slower page table lookup operation(s) a small number of times, while performing the faster read and write operations throughout the life of the application. Conversely, applications that frequently map new memory locations may suffer a performance hit with introspection, but may still be viable depending on the number of monitors working at a given time and whether their address lookups result in cache hits. The size of the LRU cache is adjustable and can be tuned to meet an application’s needs.

3.7.2 Active Monitoring Using Hooks

Hook processing is the key operation where the Turret framework will differ in performance from a traditional architecture. Therefore, our benchmark measurements look at the time required to process a single hook in the Turret framework and compare that with a traditional architecture. To measure the hook processing time with the Turret framework, we instrumented the trampoline code. We retrieved the value of the processor’s performance counter before and after the `VMCALL` instruction. The processor’s performance counter was

obtained using a function provided by Windows, `KeQueryPerformanceCounter`. The difference between these two measurements represents the time needed for inter-VM communication and hook processing within the security VM. However, this measurement is noisy. It can be influenced by cache effects, VM scheduling, physical interrupts, CPU frequency scaling, and other loads on the system. We took several steps to minimize the influence of this noise in our measurements. First, we pinned each VM to its own CPU core. Next, we disabled unnecessary services in the Security VM. Then we disabled CPU frequency scaling in the BIOS.

After the system was prepared as indicated above, we measured the hook performance across five runs, where each run included 1000 measurements. The 5000 measurements were combined into a single data set. Then we performed a standard statistical analysis to remove the outliers, since there was still some noise in the measurement. Our analysis computed the inner-quartile range (IQR) of the data set and defined outliers to be 1.5 times the IQR above the third quartile and below the first quartile. After removing the outliers, the computed mean on our data was 28 μ secs to just send a notification to the Security VM and 175 μ secs when the security application used introspection to lookup file names from handles.

To measure the hook processing time for a traditional architecture we developed a system that processes the hook inside the User VM. The kernel code in this system is the same as the Turret framework except that instead of executing the `VMCALL` instruction to process a hook, we send an event to a user-space application. This application performs the same check as our prototype, looking up the file handle to check the associated file name. The system was prepared and the tests were run the same as for the Lares architecture test. After removing the outliers, the computed mean on our data was 17 μ secs. The results from each of these tests are shown graphically in Figure 11. These graphs, known as boxplots, show the IQR as a gray box. The median value is denoted with a horizontal line through the box. And the range of the remaining nonoutlier data is shown as lines extending above

and below the box.

Two major factors contribute to the differences in the performance of Turret versus a traditional architecture. First is the fact that it takes more time to exit the User VM, send a signal to the Security VM, and perform the software address translations needed for VMI than it does to perform the same tasks locally. This factor contributes to the overhead for a single hook event. The second factor is more subtle. When everything is processed locally as in the traditional architecture, there is no security benefit to processing data inside the kernel versus in application space. Since only a subset of the calls to `NtCreateSection` are associated with the file handle of a new process execution attempt, the traditional architecture can use the result of the `ObReferenceObjectByHandle` function in the windows kernel to filter out the hook events that do not need additional processing. Using this technique, only a subset of the hook events are sent to user space for processing. However, in the Turret framework, we do not trust any local functions in the Windows kernel. So *every* hook event is sent to the Security VM for processing. This example is specific to the hook that we implemented, but a similar situation may exist for other hooks as well. This trade-off raises the mean time required for hook processing in the Turret framework, but also increases the security of our framework by reducing dependencies on untrusted code.

While our benchmarks show that Turret is slower than traditional hook processing, we also provide a significant improvement in security. The overall performance of a given application will ultimately depend on the number of hooks it uses in addition to its use of introspection and other techniques to collect data for processing each hook. Our experience with the example application and the performance results presented in this section suggest that applications using Turret can perform similarly to applications using traditional architectures.

3.8 Discussion and Future Work

One of the design goals for Turret was to enable the rapid development of new security applications. While writing a meaningful security application is never trivial, the monitoring framework should not impede the development process. In this section we provide some concrete examples of how security applications interact with the Turret framework, specifically for passive monitoring. These examples are designed to demonstrate the utility of Turret. They do not represent a complete security application. In Chapter 6, we provide an in-depth discussion of several security applications that utilize the Turret framework.

3.8.1 Using Passive Monitoring

Using introspection, XenAccess – one of the libraries provided by Turret – can view and modify data in memory of a running OS. The example below shows how to use XenAccess to view a listing of the linux kernel modules (LKM) installed in a running instance of Linux. Additional examples in the open source release of XenAccess show how to list running processes, view arbitrary memory pages, view the memory of a particular process, and dump the entire User VM’s memory into a file.

The LKM listing example uses the `xa_access_kernel_sym()` function to list the LKMs installed into the User VM kernel. This only requires 44 SLOC. Listing 1 shows the code for this example. The code follows a linked list in the User VM kernel memory using introspection. It starts by loading the memory page containing the head of the list, which is found using the `modules` kernel symbol. This address points to a `module struct`. This structure contains a circular doubly linked list that points to the rest of the modules. Therefore, the code proceeds by loading the memory page addressed by the next pointer all the way down the list. For each structure, the module name is accessed by creating a pointer to its offset, and then it is printed to `stdout`. Since the linked list is circular, the code ends when it finds a pointer back to the head of the list.

Listing 1: Source code for an example that lists all running LKMs in the User VM kernel. All error checking code has been removed for clarity.

```

xa_init_vm_id_strict(dom, &xai);
memory = xa_access_kernel_sym(&xai, "modules", &offset);
memcpy(&next_module, memory + offset, 4);
list_head = next_module;
munmap(memory, xai.page_size);
while (1){
    memory = xa_access_kernel_va(&xai, next_module, &offset);
    memcpy(&next_module, memory + offset, 4);
    if (list_head == next_module){
        break;
    }
    name = (char *) (memory + offset + 8);
    printf("%s\n", name);
    munmap(memory, xai.page_size);
}
if (memory) munmap(memory, xai.page_size);
xa_destroy(&xai);

```

Since this example is accessing and displaying OS-specific information, it requires OS-specific knowledge. In this case, the knowledge falls into two categories. First, we must know that the `modules` symbol points to the beginning of a linked list that will provide the information that we need. Second, we must know the offsets within the `module` struct needed to access information such as the `next` pointer and the module name. Requiring this type of information is common for introspection applications. For this example, the information needed was available in both the Linux source code, and Bovet and Cesati's kernel book [26]. When viewing Windows memory, both the Microsoft Windows Debugger (i.e., WinDbg) and Russinovich and Solomon's book [143] are useful references.

The example above is straightforward and provides a quick understanding of XenAccess's introspection capabilities in operation. Other monitors are not much more complex. For example, we developed an application that monitors for changes in the system call table (110 SLOC) and an application that monitors the integrity of an installed LKM (172 SLOC). The security applications of these types of monitoring are clear in areas like intrusion detection and integrity checking, and have been well explored in literature. XenAccess makes these types of applications possible by providing memory access at the proper levels

of abstraction. Based on our experience building XenAccess introspection monitors, we feel that the part of the Turret framework satisfies design goal (4).

3.8.2 Using Active Monitoring

The Turret framework provides several ways to perform active monitoring. Using hardware events, or network traffic is relatively easy. Placing hooks in the User VM kernel is more difficult, requiring kernel-level development to control the hook and the associated trampoline code. The application examples in Chapter 6 describe these techniques in greater detail. Here, we describe the process for receiving hardware event notifications within the Security VM.

After the Turret framework is installed, hardware event notifications are continuously sent to a FIFO inter-process communication channel (IPC). To receive these events, the security application must implement a FIFO server that listens for, and processes, each event. This FIFO server can be written in most programming languages, allowing for flexibility in the design of the security application. Listing 2 shows a basic fifo server for processing hardware events that is written in C.

Installing kernel hooks is more challenging. This requires the following high-level steps. First, we need to determine the correct location to place a hook. This could involve source code analysis or reverse engineering. Next, we need to determine how to properly protect the hook. In some cases this can be done by rooting the hook's protection to a hardware event, as shown in Figure 8. In other cases, it may require more complete kernel-level protections. Finally, we need to install the hook and activate the protections. Each of these steps are highly specialized, and must be performed for each new hook added to the User VM. While this is a large burden, the benefit is the ability to perform fine-grained monitoring on specific events inside the User VM.

Listing 2: Source code for receiving hardware events from the Turret framework. Each incoming event is processed in a separate function called `message_handler`.

```
FILE *fifo = NULL;
char readbuf[BUF_SIZE];
struct pollfd fds;

fifo = fopen(FIFO_FILE, "r");
fds.fd = fileno(fifo);
fds.events = POLLIN;

while (1){
    int ret = poll(&fds, 1, -1);
    if (ret > 0){
        memset(readbuf, 0, BUF_SIZE);
        if (NULL == fgets(readbuf, BUF_SIZE, fifo)){
            if (feof(fifo)){
                continue;
            }
            printf("fgets failed, exiting\n");
            break;
        }
        message_handler(readbuf);
    }
    else{
        printf("poll failed, exiting\n");
        break;
    }
}
fclose(fifo);
```

3.8.3 Future Work

The passive monitoring portions of Turret are currently well established. These techniques are implemented in the open source XenAccess Library, which has received significant use for over three years. With this in mind, the primary future work for this portion of Turret is to automate some of the setup and configuration steps, allowing for more rapid deployment, and to achieve wide scale adoption of the technology.

The active monitoring techniques are currently research prototypes. In particular, work is needed to automate the process of protecting and installing the User VM hooks. In addition, security applications would benefit from the ability to install protected hooks in user-level applications. Looking at the hardware event driven techniques, work is needed

to understand and mitigate the impact of virtualization aware hardware. As virtualization becomes more popular, hardware manufacturers are creating network card, video cards, and other devices that communicate directly with each VM instead of using an intermediary control VM. This could make it more challenging to perform monitoring based on these hardware events.

Looking further into the future, it would be interesting to consider how the Turret framework monitoring techniques could be offered as a service to VMs running under the cloud computing model. Currently, we assume that the software doing the monitoring had privileges to view all of the VMs on the platform. However, in a cloud computing model, a user may have several VMs that are running – perhaps even migrating – across multiple hypervisors. In this setting, it would be useful to be able to offer the user monitoring capabilities that are restricted to the VMs owned by the user. This could be partially addressed through the use of mandatory access control, but the general solution remains an interesting open problem.

3.9 *Summary*

Stepping back to look at the six design goals for a robust monitoring solution, we note that Turret satisfies each of these requirements. (1) The Turret framework requires only small changes to the Xen virtualization platform. These changes are used to implement memory protections and a communications channel for the hooks placed in the User VM. (2) Using the capabilities provided by Xen, the only code inserted into the User VM is the protected software hooks and the supporting trampoline code. And these minor additions are only required for applications that need this level of fine-grained active monitoring. (3) Our performance testing, discussed in Section 3.7, shows small overheads for the Turret framework components, making these capabilities effective for a variety of monitoring applications. The micro-benchmarks look promising and we demonstrate that viable security

applications can be build using Turret in Chapter 6. (4) Our example code shows that developing security applications that use Turret is straightforward, with a minimal learning curve. The one exception to this is the placement of hooks in the User VM, which requires more advanced knowledge. We noted that this is a problem that should be addressed in future work. (5) While our existing library implementation can view memory and receive event notifications, the Turret framework is easily extensible to collect any type of data from the User VM. (6) Finally, leveraging the protections provided by the hypervisor, Turret can be sufficiently isolated from the User VM, reducing the possibility of tampering by malicious software.

One of the key benefits to the Turret framework is the ability to perform active monitoring from a protected location. Active monitoring is needed to support state-of-the-art host-based security applications such as intrusion detection and anti-virus tools. However, as recent research has focused on moving security applications into an isolated VM, the resulting architectures do not support active monitoring. Turret addresses this problem by giving security tools the ability to do active monitoring while still benefiting from the increased security of an isolated VM. Our security analysis shows that Turret provides security suitable for deployment on production systems. And our performance evaluation shows that Turret's overall impact on system performance is small. The Turret framework is generally applicable to any application that requires secure active monitoring.

CHAPTER IV

IMPROVING VMI'S RESILIENCE TO SOFTWARE CHANGES

4.1 Motivation

One of the key data sources viewable using virtual machine introspection (VMI) is memory. However, the memory view available through VMI is low-level, typically just a raw view of the system's virtual address spaces. The layout of data within this memory space is dependent on the specific software running within the virtual machine, and will likely change with different versions of the software. For this reason, applications built using VMI today are often restricted to working with a specific software version and are not easily ported to other systems. This limitation has forced VMI to remain a research novelty rather than a commercial success.

The field of memory analysis is focused on addressing this problem. In general, there are three major problems to solve to enable general memory analysis applications: (1) locating data, (2) parsing data, and (3) interpreting data. Locating data involves finding where specific data structure instances are located in memory. Parsing data involves breaking the data structure into smaller pieces where each piece is a single type. And interpreting data involves determining the semantic meaning of a particular type within the data structure.

The key problem that we address in this chapter is how to locate data structures in memory. Using virtual machine introspection or a coprocessor-based monitoring solution, we cannot leverage existing APIs to provide information about the target system. For example, instead of calling a function that provides a list of the currently running processes, we must locate the data structures in memory that hold this information. Likewise, instead of calling a function that provides a list of open network connections, we must locate the data structures in memory that hold this information. After locating the data structures,

we must also be able to interpret their contents in order to extract useful information from them. In this work we focus on the task of locating the data structures while noting that the interpretation problem has been partially addressed by previous work [44].

Locating a *specific* data structure in memory is not necessarily difficult. Tools exist for locating certain data structures that have obviously unique characteristics [152]. However, not all data structures are as easy to identify. What is needed is a *general* solution to this problem. The challenge is to provide a single technique that can be used to model and identify any data structure in memory that is needed by the security application. Simple heuristics that work for specific data structures are not useful in this setting because the application developer can not be expected to identify such heuristics for each data structure that is needed. Such heuristics may not even exist for all data structures. This problem is challenging because a viable solution must be general enough to work with all data structures, yet accurate enough to correctly identify the data structures across multiple software versions.

Our solution starts by building a model for each data structure that we want to find. This model is created automatically by training the software with samples of the data structure along with samples of other portions of memory. Specifically, we use a supervised machine learning algorithm to build this model. The training set can contain multiple versions of the data structure, ensuring that the resulting model has the generality needed to operate on a wide variety of target software versions. After building the model, the security application is deployed. When the application needs to locate specific data structures on the target system, it initiates a search by testing a sample region of memory against the model. Once found, the locations of the data structures are saved into a database to allow for quick access in the future. If the target software is updated, or the data structures change locations for any other reason (e.g., restarting an application), then the database is updated by repeating the initial search procedure.

4.2 Previous Techniques

Memory analysis is a rapidly developing research area and many existing tools can provide useful information about the memory of a running system. However, the fundamental drawback to all of the existing techniques is that they lack generality. Each technique is specially constructed to work for a specific software version. Therefore, software patches can potentially incapacitate a technique, rendering it useless until it can be updated to work with the new software version. Even with this limitation, the techniques listed below represent the current state-of-the art and have enabled a wide variety of useful applications ranging from computer forensics to runtime system security.

4.2.1 Hard-Coded Values

The simplest approach to locating and parsing data structures is to hard code the location (i.e., virtual or physical address) and offset values. Often, for a given version of an operating system or application software these values are consistent on different machines and at different times. Finding the appropriate values in the first place typically involves reverse engineering, source code analysis, or vendor-provided debugging symbols. This technique is commonly used because it is easy and effective. Example applications that use hard-coded values include XenAccess [125] and Volatility [177].

4.2.2 Heuristics

A wide variety of tools have also been developed to extract specific pieces of information from memory images using basic heuristics. Schuster's PTFinder [152] finds processes and threads in Windows memory dumps by doing a linear scan of physical memory, looking for a constant pattern of bytes found in the process and thread data structures. Further work focused on finding data structures allocated in a region of Windows memory known as *pool memory*. Data structures allocated from the pool are tagged with a four byte ASCII identifier as well as size information, which allows them to be easily found in memory.

Burdach presented a technique for finding information of forensic value in memory dumps of Linux [28] and Windows [29] by using a combination of symbol information (in Linux) and heuristics (in Windows).

4.2.3 Debugger Tools

For some operating systems and applications, it is also possible to use debugging symbols provided by the software vendor to extract information about data types. Microsoft, for example, provides type information in their PDB (Program DataBase) file format. These debug symbols can be used with a number of tools supplied by Microsoft, such as WinDbg [114] and the Kernel Memory Space Analyzer [115]. However, the type information provided is not complete, so there are some data structures that cannot be found using debug symbols alone. In addition, user applications (if they provide debug symbols at all), typically distribute PDB files with type information stripped.

4.2.4 Code Analysis

FATKit, a memory analysis framework created by Petroni et al. [132], uses static analysis of application and operating system source code to build models of the data structures used. These models can then be overlaid on images of memory to provide meaningful interpretations of the raw data. This technique of extracting data types from source code was also later used by Petroni [131] to examine stored function pointers and detect malicious modifications to kernel control flow. Unfortunately, the source code of most programs and operating systems remains closed, so this technique is only useful in limited circumstances.

Another work that applies machine learning to the problem of finding data structures is the work of Cozzie et al. [44]. Their approach uses unsupervised learning to classify memory regions into potential data structures, as well as attempting to determine the types of the members of each data structure. This could be used in conjunction with our techniques in two ways: either as a post-processing step to reduce false positives produced by the Hidden Markov Model, or to aid in reverse engineering the individual fields of the data structures

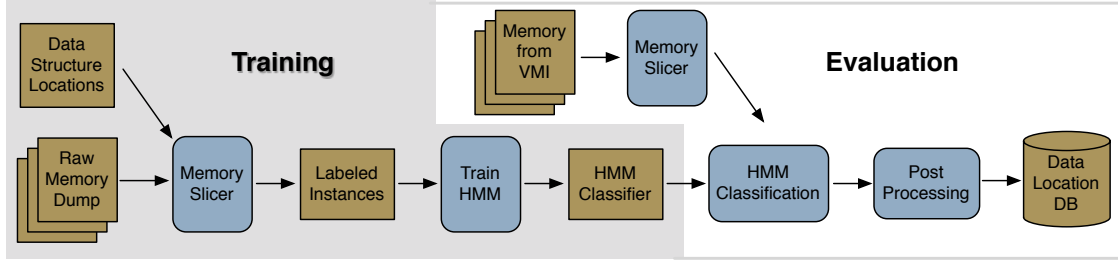


Figure 12: An overview of the major steps needed to locate data structures in memory. The training steps produce a hidden markov model (HMM) describing the data. The evaluation steps locate instances of the data structure from a new data set.

found with our method. In a similar vein, Walters describes a technique called memory informatics [178] that applies the bioinformatics-inspired techniques from protocol informatics [19] to automatically find data structures in memory. Unfortunately, the details of this method are not available, so we were unable to compare its performance with ours.

4.3 Locating Data Structures Using Machine Learning

This section describes the technical details of our approach to the problem of locating data structures in memory. Since our approach utilizes machine learning, we provide some introduction to these concepts throughout this section. We describe each step of our solution including building training data sets, building the classifier, refining the results with a post-processing step, and using the final results to identify data structures in memory. Figure 12 shows an overview of our solution.

For the purposes of describing how our system works, we will use a running example throughout this section. The running example describes how each of these steps would be used to locate the `EPROCESS` data structure from a Windows system. This is an instructive example because it highlights the main features of our system. However, the same techniques could be applied to locating any data structure within any operating system, in both kernel and application-level memory.

Our technique is best suited to data structures that are large enough to model accurately. On both Windows and Linux, most of the security critical data structures are very

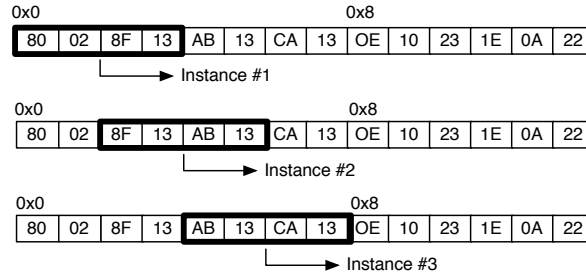


Figure 13: Slicing memory data to create instances with a sliding window. In this example, the sliding window size is 4 bytes and the interval is 2 bytes.

large. Examples from Windows include `EPROCESS` (608 bytes), `ETHREAD` (600 bytes), and `FILE_OBJECT` (112 bytes). Examples from Linux include `task_struct` (1360 bytes) and `mm_struct` (448 bytes). All of these sizes are subject to variation based on different software versions, but these values show the general size of these data structures.

4.3.1 Preparing Data For Training

The first step is to collect and label raw memory data from the systems that we want to use for training. Training on data from multiple versions of the target software allows the resulting classifier to have the generality necessary to work across a large number of target software versions. Looking at the `EPROCESS` example, we need positive samples (e.g., samples of the `EPROCESS` structure) and negative samples (e.g., samples of other areas of memory) from different versions of Windows. Finding positive samples requires knowing where data structures are located in memory. This *ground truth* information is generally obtained by reverse engineering the memory layout.

The input to the training procedure is a series of *instances* that are each composed of various *features*. We build the instances by “slicing” memory using a sliding window with a size equal to the data structure and an 8 byte interval. For example, when slicing memory for the `EPROCESS` structure on Windows XP Service Pack 2, the sliding window size is set to 608 bytes, which is the length of this data structure. The first instance is obtained from the first 608 bytes of memory (physical address 0 through 608), then the next instance starts 8 bytes later (physical address 8 through 616). We continue to create

instances throughout memory using this technique. This slicing technique is demonstrated in Figure 13. Note that even though we train on the size of the data structure in one version of the target software, our choice of classification algorithm allows us to still detect this data structure from other versions of the target software. This is true even with small numbers of insertions and deletions in the data structures. See Section 4.3.2 for more details.

Each instance is then broken into features. In order to minimize the semantic knowledge needed for parsing the features, we let each byte in the instance be a different feature with a numeric value between 0 and 255. This results in a large number of features for each instance. In order to build a model of these instances, one would typically need a huge quantity of data due to the “curse of dimensionality” [21]. In our EPROCESS example, we have 608 features that each have 256 possible values. However, our experimental results show that this system has a high accuracy, even with a relatively small training data set. We believe that this is because many of the dimensions in the data set are actually unrelated to the classification task. One could further explore this using feature selection algorithms, however our goal was to simplify the training process as much as possible and our technique of including all the data as features yields acceptable results.

After breaking each instance into single-byte features, the final data preparation step is to label the instances. Using the results of our reverse engineering effort, each instance receives a label indicating if it is the data structure that we are learning or not. These labels provide the ground truth that is used in building a model of the data.

Training data sets are prepared by slicing the memory from different versions of the target system’s software. Returning to our EPROCESS example, a training data set that includes instances from both Windows XP Professional Service Pack 2 and Windows 2000 Professional Service Pack 4 provides sufficient information to detect the EPROCESS data structure on any version of Windows 2000 or Windows XP.

4.3.2 Selecting and Training the Classifier

With the training data prepared, the next step is to create a model that can differentiate between positive and negative instances. This type of model is called a classifier. Choosing an appropriate algorithm for the classifier was challenging for this problem. Each instance is large and contains many features. And their training data files can be extremely large (e.g., tens of gigabytes). Many of the standard machine learning algorithms could not operate efficiently on this data set; often running out of memory or taking days to build the classifier.

The key insight that led toward a solution for this problem is that a data structure in memory is defined by a loose set of rules. For example, the data structure may have an `int` followed by a `char` pointer and so on. Within the context of a given data structure, each of these data types will typically have one of a small number of possible values or range of values. Combining this information for the entire data structure, one can view this data as loosely conforming to a grammar.

Hidden Markov Models Hidden Markov Models are commonly used in natural language processing because they provide a good model of a language's grammar. Viewing a data structure in memory as having its own grammar, we used the *k*-means [163] and Baum-Welch [18] algorithms to build a HMM that represents the positive instances in our training data. Since the HMM models the positive instances, it can be built using only the positive instances as input. This provides a significant benefit as it will reduce the amount of data required for training.

Another benefit of using a HMM to model the data is that it is resilient to changes in the data structure. Different software versions may exhibit minor variations, including insertion and deletion of values that may change the length of the data structure. In these cases the HMM will still be able to evaluate the data because the HMM does not restrict the size of instances that it can model. This flexibility, combined with the other benefits

mentioned above, make HMMs a good choice for modeling data structures in memory.

The k -means algorithm groups the data into initial clusters, which are used to initialize the values of the HMM. The clusters are initially chosen randomly, and then refined based on the natural centroid locations in the data. Since we are only using k -means to determine initial values for the HMM, we only perform one iteration of this refinement process and do not need to wait for convergence. The input to the k -means algorithm is the positive instances from our training data set and the value for k , which determined the number of clusters to use. We choose $k = 50$ experimentally using both cross-validation and performance metrics to find a suitable value. The output from this step is an initialized HMM.

The Baum-Welch algorithm starts with this initialized HMM and refines it to be more accurate using multiple iterations. Baum-Welch uses a dynamic programming approach known as the forward-backward algorithm to calculate the probability of observation sequences based on the current HMM. The HMM is refined through a generalized expectation-maximization algorithm that iteratively refines the HMM parameters to find a more optimal HMM. Given that the probabilities for a given observation sequence can be small, and given that we are working with long observations sequences, we used a modified version of Baum-Welch that uses scaling to prevent underflows in the probability calculations [135]. The output from the Baum-Welch algorithm is an HMM that describes the positive instances from our training data set.

The HMM that we create using this technique is very accurate. Our evaluation results show a false positive rate around 2.3% and a false negative rate near 0% (see Section 4.3.5 for a complete discussion of our results). However, memory data is highly unbalanced. For example, a training data set may contain tens of thousands of negative instances and less than 50 positive instances. Given this unbalanced data, we can have more false positives than true positives when using only the HMM classifier. The solution is to use a post-processing step during the evaluation, as described in Section 4.3.3. This post-processing

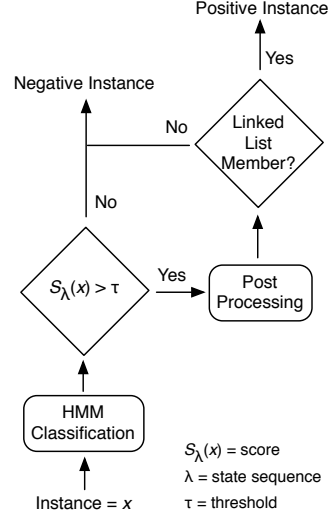


Figure 14: The evaluation steps for unknown instances. Here we show the linked list post-processing algorithm, but other post-processing algorithms may be used as well.

step further reduces the false positive rate. In the case of the EPROCESS data structure, this post processing step reduces the false positive rate to around 0.103% as described in Section 4.3.5.

4.3.3 Classifying Unknown Data

After the training is complete, we have a classifier that can be used for classification, which requires data to be prepared the same as it was for training, with the single exception that the data does not need to be labeled. Using the HMM classifier we can determine which instances from the data set are samples of the data structure that we wish to locate. The remainder of this section describes the details of how the evaluation process works.

Using the HMM Classifier The data is first evaluated using the HMM classifier. While we refer to it as a classifier, the HMM by itself is not a classifier. Instead, it is a description of a HMM that represents the positive instances. To use the HMM as a classifier, we use the following steps:

- Determine a numerical score for each instance indicating the likelihood that the given

instance can be described by the HMM. This score is determined as part of our evaluation and is described in more detail below.

- The instances with higher scores are likely the data structure represented by the HMM. The instances with lower scores are likely not the data structure represented by the HMM. Determine a threshold value to separate these two classes of instances.
- Using the threshold value, classify each instance as positive (score is above the threshold) or negative (score is below the threshold).

We calculate the numerical score for each instance using a two step process. First, we determine the most likely state sequence in the HMM for a given instance using the Viterbi algorithm [170]. For the second step, we could calculate the combined probability of the state transitions in this sequence to use as the score, but this has some limitations. This combined probability, P_λ , would be computed with the following equation:

$$P_\lambda = \prod_{i,j \in \lambda} A_{ij} q_{obs,i} \quad (1)$$

where A_{ij} is the transition probability from state i to state j , q_{obs} is the emission probability of observation obs in state j , and λ is the most likely state sequence computed above. Using this equation, if there is a single state along the path that has a 0% emission probability for the given observation, then P_λ would equal zero. This scenario is plausible given that our training data may be incomplete (e.g., the training data may not enumerate all possible options for all variables in the data structure). However, a score of zero is not useful for classification purposes.

To avoid this problem we define the score, S_λ , with the following equation:

$$S_\lambda = \sum_{i,j \in \lambda} A_{ij} q_{obs,i} \quad (2)$$

Note that the probability calculation for each state remains the same, but now we compute the sum of each probability instead of the product to obtain the overall score. This

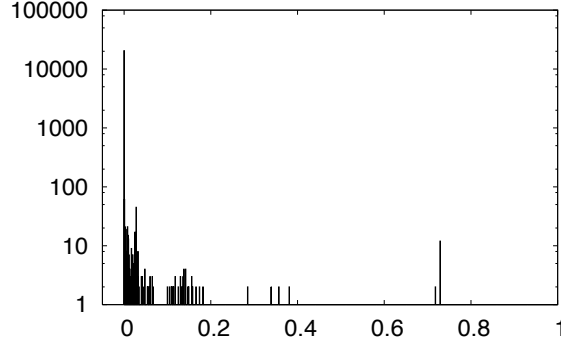


Figure 15: A histogram showing the distribution of instance score values for an evaluation of the EPROCESS data structure. The bar heights are represented on a logarithmic scale due to the unbalanced nature of the scores.

score calculation avoids the problem outlined above, while still allowing high probability instances to have a higher score.

After scoring each instance, we must determine which instances should be classified as positive and which instances should be classified as negative. This decision requires establishing a threshold value that will be used to separate the two classes of data. Figure 15 shows that the distribution of scores is lopsided. This is to be expected because the data set is unbalanced. With this in mind, the threshold value should be determined such that it will classify the large number of low scores as negative. For this reason, we compute the mean score value and use this for the threshold, τ :

$$\tau = \frac{1}{n} \sum_{i=1}^n S_{\lambda}(x_i) \quad (3)$$

In this equation n is the total number of instances in the data set and x_i is the i^{th} instance. Intuitively, we can see that the mean score value will be weighted toward the smaller values in the distribution given the large number of smaller values. This allows for the values in the tail of the distribution to be included in the positive instances. Our empirical results support that the mean is a good threshold value, as discussed in Section 4.3.5.

For many categories of data structures, we can use a post processing step to reduce the number of false positives, as described in Section 4.3.3. For other data structures, the results can be used directly as described in Section 4.3.4.

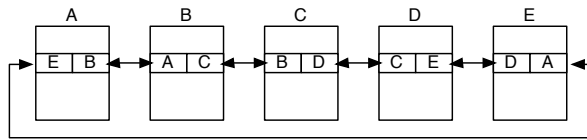


Figure 16: Data structures connected by a circular, doubly-linked list. Note the invariant that each previous pointer has a matching next pointer, and each next pointer has a matching previous pointer.

Post-Processing Post-processing is useful when we know an invariant about the data structure that we are trying to locate. One common invariant with data structures is that many are members of a linked list. Other invariants may include specific header information, pointers to shared data structures, or constant-valued data. In this section we focus on linked list detection because this is quite common and the detection technique can work across a variety of linked list implementations.

Our linked list detection will work for any doubly linked list implementation where the next and previous pointer values are stored adjacent to each other in memory. This is a common implementation technique and it is seen in the Windows kernel, the Linux kernel, and a large variety of application-level software. The invariant that we exploit is that for each next pointer there should be exactly one previous pointer with the same value. Likewise, for each previous pointer, there should be exactly one next pointer with the same value. This invariant is strictly true for circular, doubly linked lists as shown in Figure 16. However, a minor alteration to the algorithm can detect non-circular lists as well.

Our post-processing algorithm for linked list detection works without any prior knowledge beyond what is described above. It does not need to perform any address translations, nor does it need to know where in the data structure the previous and next pointers are located. The algorithm works in two stages. First it identifies the location of the previous and next pointers. Next it determines which instances are contained in a linked list at those pointer locations.

To find the location of the previous and next pointers, we test each possible location in the data structure. For each offset within the data structure, we test how large of a linked list

could be formed if that offset were the correct location for the linked list pointers. This test is done by searching for the number of instances that satisfy the invariant. To ensure that the items we find do form a single linked list, we follow the `flink` values and ensure that this traversal touches each instance in the list. To avoid performing address translations, we do not completely resolve the `flink` pointer values. Instead, we match the last 12 bits of the pointer address with a corresponding value in the set of potential list items. This works due to the semantics of virtual addresses on the x86 architecture. We assume that the offset that forms the largest linked list after this verification step is the correct offset.

After we have determined the correct offset, we search all of the positive instances from the classifier-based evaluation to see which ones satisfy the invariant. The instances that form a linked list are assumed to remain positive and we change the classification of the others. The net result of this post processing step is a reduction in false positives. Similar post process procedures could be derived for other invariants, based on the type of data structures that are being located.

4.3.4 Using The Results

After the classification and post-processing steps are complete we have a list of positive instances. In practice, this is a list of the physical address locations of the data structures that we need to operate a security tool. Most security applications will access these data structures frequently to perform integrity checks and to identify malicious activity. However, as discussed in Section 4.3.5, the process of finding each of these data structures takes some time to complete.

The solution to this problem is to store the results from the costly classification and post-processing operations into a database where they can be accessed quickly, as needed. This solution works because data structures tend to be initialized in one location in memory, and then stay in that location until the system is turned off. In fact, the memory locations are often the same after a system reboots and reinitializes its data structures if the system

software has not been updated. Using this strategy, we suggest the following initialization procedure for security tools using memory analysis:

- At startup, check the database for any pre-computed data structure locations. If locations exist, use them. If locations do not exist, then run the classification and post-processing steps to initialize the locations in the database.
- If at any time the locations taken from the database do not work (e.g., the security tool is unable to obtain the correct information from these locations), then repeat the initialization procedure and save the new results into the database.
- *Optional:* The classification models and post-processing algorithms can be updated as needed to refine these steps or to add support for new major software versions. For example, a single model may work for all versions of Windows 2000, Windows XP and Windows Vista, but may need to be updated for future major releases of the operating system. These updated models can likely be trained on pre-release versions of the software, ensuring that the security tools will provide proper coverage as soon as the new software is released.

Using this strategy, our techniques for locating security critical data structures can be integrated into memory analysis based security tools, providing a straightforward way for these tools to work across multiple target software versions.

4.3.5 Accuracy and Performance Evaluation

We performed experiments on memory images obtained from different versions of Microsoft Windows, as described in Section 4.3.5.1. We identified three data structures that provide data from the operating system kernel to use for our testing. These data structures provide information about the processes (EPROCESS) and threads (ETHREAD) running on the system, and open files (FILE_OBJECT).

Table 1: Accuracy of our technique for determining the location of data structures in memory. This table shows results for the HMM classification used alone.

	HMM Classification		
	<i>Accuracy</i>	<i>False Positive</i>	<i>False Negative</i>
EPROCESS	97.891%	2.112%	0.000%
ETHREAD	98.666%	1.337%	0.068%
FILE_OBJECT	96.606%	3.401%	0.000%

Table 2: Accuracy of our technique for determining the location of data structures in memory. This table shows results for the HMM classification used in conjunction with post-processing that identifies data structures that are members of a linked list.

	With Post-Processing		
	<i>Accuracy</i>	<i>False Positive</i>	<i>False Negative</i>
EPROCESS	99.894%	0.103%	3.125%
ETHREAD	n/a	n/a	n/a
FILE_OBJECT	n/a	n/a	n/a

For each of these data structures we performed the training and evaluation steps to measure how accurately the classification and post-processing techniques can identify the data structures in memory. We also measured the time needed to complete each of these steps in order to show the performance characteristics of the system. The remainder of this section provides details about our accuracy and performance evaluation.

4.3.5.1 Accuracy

Our data sets include memory images from a variety of versions of Microsoft Windows as enumerated below:

1. Windows 2000 Professional (no service pack)
2. Windows 2000 Professional (Service Pack 4)
3. Windows XP Professional (Service Pack 2), image 1
4. Windows XP Professional (Service Pack 2), image 2

5. Windows XP Professional (Service Pack 2), image 3

We worked with one memory image for each of the different Windows 2000 versions, and three memory images for the Windows XP version. We built each training data set using two different images. The first training data set, `train1`, was created using data sets (2) and (3). The other training data set, `train2`, was created using data sets (3) and (4). Then we evaluated the accuracy of both `train1` and `train2` using data sets (1) and (5). The end result was two sets of accuracy scores for each data set; one using `train1` and one using `train2`. Table 1 shows the average of the results from each of the test runs for each data structure.

For each data set, we must determine the ground truth information. To determine the ground truth information, we started with “clean” data sets (i.e., data sets acquired from a fresh installation of the operating system that is known to be free of malware). Next, we used existing memory analysis tools to identify the data structure locations [177, 152]. These tools rely on heuristics that were determined through reverse engineering. They are well suited to determining ground truth information for specific software versions, but they do not provide the generality that our technique provides. We used the information from these tools to label the data set as described in Section 4.3.1. The labels are used for training and for measuring the accuracy of the evaluation steps.

The results in Table 1 show that the HMM classification step is able to consistently and correctly classify a large portion of the data set as negative instances while maintaining a 0.0% false negative rate. Our post processing step shown, which focused on linked list detection, was able to significantly reduce the false positive rate although it occasionally introduced a small number of false negatives as shown in Table 2. The `ETHREAD` and `FILE_OBJECT` data structures are not part of a single linked list, so we did not perform any post processing on them.

In Section 4.3.4 we discussed how the classification results can be used in a security application. With this usage scenario, it is important to consider the impact of false positives

on the application. A false positive indicates that we have falsely identified a memory location as being an instance of the data structure. The security tool can handle this information in a variety of ways. For certain data structures, the security tool can discard the false positives after performing some additional analysis on the data. For example, if an EPROCESS data structure does not have the proper values in its fields (e.g., no process name, or no pointer to a valid handle table), then it can be safely assumed to be a false positive. Future research could help to identify the exact information that must be in a given data structure in order for it to be valid. These invariants could then be used to make this filtering process more robust.

In other cases, there may be no way to remove the false positives. However, for many security applications this will have a minimal impact. For example, it may simply result in the application performing extra security checks. The false negative metric is more important for most security applications. Our false negative rate is very low, ensuring that the correct data structures are identified and that attacks on the target system will not be missed.

For applications that require perfect accuracy (no false positives and no false negatives), our system is still a valuable tool. Previously, the reverse engineering effort to locate each data structure required analyzing a huge quantity of information to find a few instances of the data structure in memory. However, in addition to being useful at runtime for security applications, our technique can be useful to assist engineers in locating the data structures in memory. Instead of following a series of pointers or searching for the data structures across a large space in memory, engineers can use our tool to direct their search to a few specific memory locations.

While our experiments focus on different versions of Windows kernel memory, the same techniques will find data structures in memory from applications and other operating systems. This is because our technique makes no assumptions about the memory. The HMM is trained to identify sequences of bytes, and any piece of software will represent its

data using bytes in memory. This invariant holds regardless of the language the software was written in, the libraries used to build the application, or the operating system running the application. Furthermore, this invariant holds for systems running in virtual machines or systems running directly on the hardware. Finally, this invariant holds for a wide variety of computer architectures including x86, PowerPC, and SPARC.

4.3.5.2 Performance

We performed all of the training and evaluation on a system with a 2.33 GHz quad-core processor with 4 GB of DDR3 1333 MHz memory. Since the evaluation calculates the score for each instance independently, the evaluation steps could be performed in parallel. Our evaluation routine divided the data set into four parts, and then evaluated each part on a different processor core. This parallelization provided approximately a 2-times speedup for the overall evaluation performance.

We measured the time to complete the training, evaluation, and post-processing steps for each of our experiments. Training is very quick because it operates only on positive instances. The training step completed in an average of 16 seconds. Evaluation requires an average of 58 milliseconds per instance, and post-processing requires an average of 425 milliseconds per instance. Note that post-processing only occurs on the positive instances, so the overall time required for this step is small. The typical snapshot of Windows kernel memory contains approximately 100 MB of data, which is approximately 1.3×10^7 instances. Therefore, analyzing the entire kernel memory space takes approximately 4 days.

The time needed to analyze the kernel space should be viewed as a worst case scenario. Many applications use much less memory than the kernel, so evaluation would be considerably quicker for these cases. Either way, it is important to note that this evaluation step is only performed when discovering the data structure locations for the first time. The typical runtime scenario for security applications using memory analysis would be to access the data locations from a database, as discussed in Section 4.3.4.

4.4 Discussion and Future Work

VMI relies heavily on memory analysis. And current memory analysis techniques are brittle, unable to survive a software upgrade on the target system. The work presented in this chapter is a first step toward addressing this problem through memory analysis techniques that are sufficiently general to work across multiple software versions, while still being precise enough for practical use. However, this work is still preliminary. Additional testing, analysis, and refinement are needed to fully understand the applicability of these algorithms.

Additional testing would use more datasets to verify the algorithms performance across different software versions. This could include various major versions, service pack levels, and patch levels for Windows. It could also include other operating systems such as Linux, Mac OS X, BSD, Solaris, etc. And it could include application-level software on all of these systems. For each of these datasets, one could perform additional analysis. This analysis could consider the algorithm performance for different data structures to determine if accuracy is influenced by factors such as data structure size, type, or entropy. Finally, algorithmic refinement would look at all stages of the algorithm – but especially the initialization and post-processing steps – to identify a technique that works well across all datasets.

Even with this additional refinement, since this is a machine learning approach, we can never expect to achieve perfect accuracy. For some security applications, this is a problem. In these cases, the only solution would be to use memory analysis techniques that are error-free. The two primary options in this domain are using static analysis of the source code or vendor-provided symbolic information to interpret data in memory. However, some security applications and many non-security applications can provide useful results without perfect accuracy. In these cases, or when source code or symbolic information is not available, we believe that a machine learning approach offers a valuable alternative to the current state-of-the-art.

4.5 *Summary*

Memory analysis techniques are critical to the success of new security application architectures that use VMI. These techniques are also required to detect advanced malware that operates entirely within memory, without ever writing to disk. The work presented in this chapter provides a starting point towards addressing the issue of locating security critical data structures in memory, which is one of the key challenges facing memory analysis applications today. Unlike previous work in this space, our system aims to work across a wide variety of data structures, types of software, and operating systems. Properly trained models will locate a data structure, even when the target system has received software patches that make changes to the data structure.

CHAPTER V

GYRUS: A VMI-BASED SECURITY FRAMEWORK

5.1 *Motivation*

Computers are often compromised and then used as computing resources by attackers to carry out malicious activities such as DDoS, spam and click fraud. Distinguishing between network traffic resulting from legitimate user activities and illegitimate malware activities is a very challenging problem because many of the activities performed by modern malware (e.g., bots) are similar to activities performed by users on their desktop computers. For example, users send email and malware sends spam; users view web pages and malware commits click-fraud; and instead of using a customized or a rarely used protocol that would arouse suspicion, malware is known to tunnel any malicious traffic through commonly used protocols such as HTTP to give it the appearance of legitimate application traffic. This can be accomplished by the malware running the application protocol or injecting itself into a legitimate application. Furthermore, malware can mimic user activity patterns, such as time-of-day and frequency, and can morph and change tactics in response to detection heuristics and methods to hide its malicious activities and traffic.

As a result, existing security technologies such as firewalls, anti-virus, intrusion detection and prevention systems, and even botnet detection systems such as BotHunter [77] and BotMiner [76], all fail or have a significant capability gap in detecting and stopping malicious traffic, particularly where it is disguised as legitimate application traffic. For example, host-based application firewalls allow traffic from legitimate applications and thus cannot stop malicious traffic from malware that has injected itself into a legitimate application. Previous research projects such as BINDER [45] and Not-A-Bot [78] aimed at distinguishing user-intended network traffic based on timing information of user input lack

the precision necessary to identify traffic created by malicious code injected into a legitimate process and sent shortly after a user input event.

Since malware or bots are not human users by definition, their traffic is not directly initiated by user activities on a host. Thus, if we can have the infallible observation that the traffic is not user initiated, or in other words, that the data is not what the user has intended or authorized the application to send, we can detect it as malicious traffic and stop it from leaving the host. Based on this insight, we propose a new approach to detect and stop malicious traffic that is disguised as legitimate application traffic. Our framework, named Gyrus¹, is an efficient and robust approach based on virtual machine introspection techniques that use hardware events combined with memory analysis to authorize outgoing application traffic only if the data was really what the user on the system intended the application to send. With Gyrus, a host can prevent malware from misusing networked applications to send malicious traffic even if the malware runs an application protocol correctly or injects itself into a legitimate application.

Our approach is based on the simple premise that malware can not reproduce hardware events coming from the keyboard or the mouse. Building from these events, our Gyrus framework interprets a user's intent based on his interactions with the desktop computer and the semantics of the application that receives the user inputs. This intent is then dynamically encapsulated into a security authorization that the framework uses to distinguish legitimate user-initiated application traffic from illegitimate malware-initiated traffic that appears to be from the application.

Gyrus is by design application-aware. That is, for each networked application (e.g., email and web browsing) that may be misused by malware to send and hide malicious traffic, Gyrus understands the semantics of that application's user input and how this input maps to data that the user intends the application to send out. Gyrus also understands the cases where the application is allowed to automatically send traffic (e.g., sending previously

¹The fusiform gyrus is a part of the human brain that performs face and body recognition.

composed messages or auto-fetching or refreshing a web page) that has been implicitly authorized due to previous user actions or application start-up or configuration activities (see Section 6.2 for more details). In other words, Gyrus provides the ground truth information that links user intent with the observed application traffic from a host. This information can then be used to facilitate a wide variety of security policies, often in conjunction with existing security technologies. For example, in a high security setting with a well known and restricted software installation base (e.g., a bank or government), Gyrus could be used in conjunction with whitelisting, firewalls, and intrusion prevention systems to prevent unintended network traffic disguised as legitimate application traffic from leaving the network security perimeter. In this case, Gyrus would require application knowledge for host applications that use the network in response to user input, as described in Section 6.2. For home users, or other low security settings, Gyrus with built-in knowledge of the most commonly used networked applications (such as email, instant messaging, and web browsing) could be used to filter the outgoing network traffic in these application protocols to stop the common channels of malicious traffic such as spam, click fraud and tunneled traffic, significantly reducing a compromised machine's overall utility to malware.

The primary contributions to this thesis include (1) the Gyrus framework that enables secure monitoring of user interactions with applications on the host and detecting and stopping malicious traffic from leaving the host disguised as legitimate application traffic, based on the key insight that malicious traffic is sent by malware without user consent whereas legitimate traffic is typically initiated by a user input or command; (2) the use of virtual machine introspection and memory analysis to precisely determine the expected application behavior based on user input events, in particular, what traffic data the user intends the application to send out; and (3) the demonstration of the viability of Gyrus by implementing the framework along with support for applications in Windows XP, as described in Chapter 6. Our prototype currently supports email and web browsing and can be extended

to include instant messaging and voice-over-IP, effectively covering the most common network applications.

The rest of this chapter is organized as follows. Section 5.2 discusses previous research efforts with goals that were similar to Gyrus. Section 5.3 discusses the security requirements for any framework driven by hardware events. This section also presents our threat model and security assumptions. Section 5.4 introduces the Gyrus framework, which is followed by a description of our implementation in Section 5.5. Then, building on the Gyrus framework, Chapter 6 details our experiences using Gyrus with email clients and web browsers to combat spam and click fraud, respectively.

5.2 *Previous Techniques*

Gyrus is most closely related to previous work on identifying human intent. It is also closely related to a variety of host and network security technologies in that they share similar high-level goals. We discuss each of these areas below to show how Gyrus improves on the current state-of-the-art and to provide some background context for our work.

5.2.1 Identifying Human Intent

Gyrus shares the goals of two previous research projects: BINDER [45] and Not-A-Bot [78]. Both of these projects aimed to identify the user’s intent and then use that knowledge to block or classify network traffic. BINDER’s key assumption was that “outbound network connections made by a process that receives user input a short time ago is user intended”. Building on this assumption, the only technique used to determine user intent in BINDER was the timing between user input to a process and network output from that process. Malicious software can easily defeat such systems by injecting code into an existing system process and sending network traffic shortly after that process receives user input. Another drawback to BINDER is that the system was deployed within the user’s operating system, leaving it vulnerable to attack by malicious software. The Not-A-Bot (NAB) [78] project builds on BINDER using code protected by a Trusted Platform Module (TPM) [75] to mark

outgoing traffic as human-generated if it occurs soon after a hardware input event. NAB suggested that this packet marking could be used to give priority to user generated traffic on the network.

Both BINDER and NAB determine user intent using only timing information. Gyrus is more discriminating, using application knowledge to determine the precise traffic that results from user input events. BINDER does not protect its software from attack, whereas Gyrus uses a virtualization-based architecture to provide a trusted execution environment. NAB requires changes to all user applications to support attestation of network traffic; by contrast, the Gyrus framework works with existing commodity operating systems and applications, improving host security without invasive modifications. Finally, because both BINDER and NAB use the recency of hardware input to decide if traffic is legitimate, they cannot handle asynchronous network traffic, such as an email that is queued while the network is down and sent much later. Gyrus has no such limitations. Instead, content-specific authorizations are created when the user initiates the action that will result in a network send.

Another related system, Siren [25], injects specially crafted synthetic user input into an idle operating system. This input corresponds with a well defined network output, allowing the system to monitor for any unexpected network output. Siren's goal is to identify malicious software that hides its network output by sending at nearly the same time as regular user output. However, the synthetic user input can manipulate the computer's state when, for example, a user steps away for a short period of time. Gyrus is able to achieve the same security goals without these usability drawbacks.

Other security systems have attempted to distinguish human actions from those of malware. The most widely known of these is CAPTCHA [174], which challenges the user to prove that she is human by solving hard computer vision problems. This approach is inappropriate in our context, however, as it would require explicit interaction to authorize a given network event. Our solution, by contrast, *implicitly* approves human-generated

network traffic by linking it to the hardware events that generated it.

5.2.2 Detecting and Containing Malicious Activity

Prior work on malware detection and containment is largely complementary to our work. We do not claim that Gyrus should replace existing security infrastructure such as firewalls, whitelisting, intrusion detection systems, and anti-virus systems. Instead, we believe that Gyrus augments these systems to improve one’s overall security posture by linking user intent with observed network traffic.

Traditional network firewalls [35] filter traffic based on Internet protocol (IP) addresses, network ports, or other network-level attributes. Application firewalls such as ZoneAlarm [2] or VMWall [162] can also filter traffic based on which process it came from. Whitelisting can be useful in creating appropriate firewall policies such as limiting the applications that can send traffic [1] or identifying authorized recurring traffic signatures [23]. Gyrus augments these capabilities by providing a framework to filter traffic based on a user’s interactions with applications.

Gyrus also complements other security infrastructure including intrusion detection systems (IDS) [124, 63], intrusion prevention systems (IPS) [154, 153], and anti-virus software [165]. IDS and IPS systems represent a broad class of tools that detect and prevent malicious activity on networks. These systems use a variety of detection algorithms ranging from signature to anomaly detection [11], and frequently use statistical or machine learning approaches to combine multiple detection features [106]. Beyond BINDER and Not-A-Bot, we are unaware of any IDS or IPS system that incorporates user intent as a feature in its analysis. Our framework could be used to improve these systems through the addition of this information. Finally, given that anti-virus software is known to have incomplete detection capabilities [122], Gyrus can help to provide a defense in depth strategy [119] by detecting malware through its outgoing network traffic.

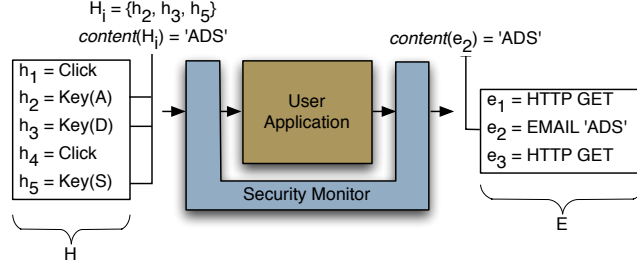


Figure 17: The formal requirements for a security framework driven by hardware events. A sequence of hardware events H results in a set of network protocol events E . In particular, the subsequence $H_i = \{h_2, h_3, h_5\}$ causes an email to be sent (e_2). The security monitor intercepts hardware and network events.

5.3 Requirements and Threat Model

In designing Gyrus, our goal was to build a framework that is applicable to a wide variety of realistic deployment scenarios. This section describes the specific high level requirements, along with the threat model and assumptions that we used in designing our framework.

5.3.1 Requirements

Our goal is to identify network traffic that results from human activity on the host. More formally, we have a set E of monitored network protocol events, such as HTTP GET requests and SMTP email sends, and a sequence H of hardware input events, such as key presses and mouse clicks.

For each $e \in E$, we must decide whether it was generated by human input or by automated means. This is equivalent to asking if there is some subsequence $H_i \subset H$ such that:

$$H_i \rightarrow e \quad (4)$$

and

$$\text{content}(H_i) = f(\text{content}(e)) \quad (5)$$

That is, (4) e was sent as a result of H_i , and (5) the *content* produced by the hardware events in H_i is the same as the content observed in e , modulo some well-defined transformation f (e.g., adding headers or applying compression). Informally, to permit a particular

event e , we must establish that it was caused by H_i and that its content is the same as what the user entered. Figure 17 shows an example of how these values are set as a user interacts with his computer.

In practice, a security application may not be able to satisfy these precise requirements. For example, in the Not-A-Bot paper Gummadi et al. [78] note that in order to validate the content of an email based on keystrokes, one must find a matching subsequence from all observed key presses and match it against the version seen on the network. Moreover, a simple subsequence is not sufficient. Clicking the mouse might move the cursor back to a previous point in the message, causing subsequent keyboard input to appear earlier in the outgoing email.

Our approach sidesteps the problem of directly validating outgoing content based on hardware input by assuming that it is possible to securely view the current state of the machine using *introspection*. This affords us a richer view of the user's inputs, and allows us to create an approximate binding between observed network traffic and user action.

5.3.2 Threat Model and Assumptions

In order to provide assurance that network traffic leaving the host is human-generated, we must make several assumptions. First, we assume that our system sees hardware events before the monitored OS, and that such events cannot be forged by an attacker. Second, in order to assure the integrity of our security software, we assume that it has a trusted execution environment (such as that provided by a secure virtualization environment or a platform implementing trusted computing [161]) that is isolated from malicious software running in the user environment. We chose to use a virtualization-based environment for these purposes. The Trusted Computing Base (TCB) for our system is composed of the hypervisor and all of the software in the Security Virtual Machine (VM), which includes the Gyrus framework.

Finally, because our solution uses virtual machine introspection (VMI) to extract user-generated content and determine the meaning of hardware input events (e.g., to distinguish a click on the send button of an email client from clicks elsewhere), we must assume that the layout of the operating system (OS) and application-level data structures we examine have not been altered. This assumption, which we call the *introspection assumption*, is common to most current VMI-based solutions [69, 91, 126]. It also represents a fairly high bar for the attacker because modifying the layout of these data structures would require updating all code in the system that uses them. Otherwise the affected OS or application would no longer function properly. These updates would be challenging to perform and to hide.

In our threat model, we assume that the attacker can compromise the user’s OS and any application running inside it. Aside from the restrictions implied by the introspection assumption, he is free to execute arbitrary code, but cannot violate the security isolation provided by the trusted execution environment. We also assume that the attacker does not have physical access to the host, and hence cannot manipulate hardware events before they reach the guest OS. Finally, we assume that the attacker cannot make modifications to the hardware (e.g., by flashing firmware, in order to generate fake hardware inputs). While Chen did demonstrate the feasibility of such firmware attacks [32], they are easy to prevent by requiring signed firmware updates. Furthermore, this attack is only viable on select high-end devices with reprogrammable firmware. These assumptions reflect a realistic attacker who has successfully infected a user’s system with malware and obtained administrative privileges.

5.4 The Gyrus Framework

The Gyrus framework design requires three key functionalities in the underlying system: the ability to interpose on any hardware events of interest (see below for details, but at least keyboard and mouse events are required), the ability to view the user system’s memory, and

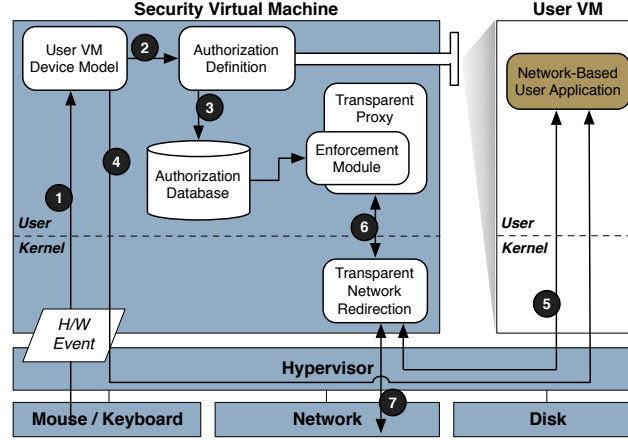


Figure 18: A hardware input event (1) starts the process of adding an authorization to the database (2-3). After the event is processed by Gyrus, it is sent to the User VM (4). Finally, a process in the User VM generates network traffic (5) that is approved or denied based on authorizations in the Gyrus database (6-7).

the ability to interpose on the user system’s network traffic. In addition, the system must provide these functionalities in a secure execution environment. Therefore, with proper hardware support, the Gyrus framework could be implemented using virtualization, a co-processor card, or a TPM-based system. For simplicity of presentation, in this thesis we focus solely on the framework design for a virtualized environment.

Figure 18 shows the major components of our framework. It is designed to leverage a virtualized environment where the security components reside in one virtual machine and the user performs his everyday work in another virtual machine. This configuration has been commonly used in recent security research [69, 91, 126] and provides the benefit of controlled isolation between the User VM and the Security VM. One key aspect of Gyrus is that there is no need to modify any software in the User VM. Because the framework’s software is in the Security VM and virtualization provides isolation between the VMs, it is very difficult for an attacker to compromise the security of the framework. Chapter 6 provides a more thorough analysis of Gyrus’ security.

The framework is completely driven by hardware events. Looking in Figure 18, the initial event comes directly from the hardware (1). Note that the figure only shows keyboard

and mouse events for simplicity, but this initial event could also come from the network, disk, or any other piece of hardware. This event comes to the Security VM, and the framework performs several actions before sending it to the User VM. These actions identify if the event is one that we care about, and if it is then the framework performs application-specific memory analysis using VMI to create an authorization (2). The authorization is placed in a database (3) and the event is passed to the User VM (4).

After sending the initial hardware event to the User VM, the framework waits to receive a network event from the User VM. We ensure that no modifications are needed within the User VM by using transparent network redirection to send any network traffic that requires an enforcement check to a transparent proxy (5). This network redirection allows the framework to inspect the network traffic without any configuration changes, software patches, or any other modifications in the User VM. When the network traffic reaches the transparent proxy, the framework searches the database for an authorization matching the network traffic (6). If an authorization exists, then the network traffic is allowed to reach the external network and the user's application works as expected (7). However, if there is no authorization, then the network traffic is rejected.

The framework tasks can be grouped into three major components: hardware event interposition, dynamic authorization creation, and enforcement. We provide more details on each of these framework components below.

5.4.1 Hardware Event Interposition

As stated above, the Gyrus framework requires the ability to interpose on hardware events. Virtualization has a similar requirement because these events must be multiplexed between the virtual machines running on a single computer. Building off of this invariant, the framework taps into the location where this multiplexing occurs. This ensures that Gyrus has access to every hardware event. This mechanism amounts to a keystroke (and mouse event)

logger running in the Security VM. However, unlike many keystroke loggers that are designed with malicious intent, we utilize knowledge of these events to improve the system's security.

It is important to emphasize that the hardware events received by the framework come *directly* from the hardware without passing through the User VM first. In most virtualization environments, this is handled in one of two ways. In Type I virtualization (where a hypervisor runs directly on the hardware), the hardware interrupts go directly to the hypervisor and then they are either multiplexed from within the hypervisor or passed to a special virtual machine that multiplexes the events. In Type II virtualization (where a host operating system runs directly on the hardware), the host operating system receives the hardware interrupts and then multiplexes them to the virtual machines that are simply running as processes within the host operating system. Either way, the key point is that these hardware interrupts are received by the framework before being received by the User VM. This means that malicious software in the User VM will not be able to forge or modify any hardware events. Gyrus builds on this property to ensure the overall security of the system.

5.4.2 Dynamic Authorization Creation

The most complex part of the framework is dynamically creating an authorization for a network event, given the initial hardware event. Although this process is application-dependent, Gyrus follows the same high-level steps for each application. The first step is to determine if the particular hardware event in question is one that Gyrus is interested in. For example, we are only interested in events that generate network traffic that we want to control. For some applications, network traffic may be generated by a particular keystroke (e.g., pressing the ENTER key), a key combination (e.g., pressing the CTRL key and the ENTER key at the same time), or by clicking the mouse in a particular location (e.g., clicking on a button to send an email message). For keyboard events, this check can be performed by analyzing the particular keystroke in question along with which application

is in focus in the User VM. The in focus application is the program that the window manager is currently sending keystroke events to, and can be determined through analysis of the user operating system's memory state using VMI. For mouse events, this check can be performed by analyzing the mouse event's coordinates, the application window on top at the coordinates, and the user interface (UI) widget located under the coordinates. Again, this information can be obtained using VMI. This step requires specific knowledge of the user operating system's window manager. We provide further details of these memory analysis steps in Section 5.5.

Once we know that the hardware event is going to trigger a network event that we need to authorize, the next step is to create the authorization. Ideally, the authorization should be as specific as possible in order to prevent malware from benefiting from it. For example, an authorization that simply allows one email message to be sent whenever a user clicks on the UI component to send an email is not sufficient. In this example, malware could use that authorization to send its own email before the user's message is sent. By making the authorization more specific, such problems can be avoided. In the case of the email example, instead of allowing any email to be sent, we should only allow an email with a specific recipient, subject, and message body. Gyrus needs an *authorization creation module* to provide the application-specific network authorization for each application that it supports. The goal of this module is to create a precise authorization using any of the information available to it (e.g., introspection, network traffic, storage device contents, and video card frame buffers).

Once the authorization is created, it is placed into a database where it can be retrieved at any time to validate outgoing network traffic from the User VM. In some cases, Gyrus may need to leave each authorization in the database indefinitely whereas in other cases it may remove the authorizations after use or after a period of time. Such decisions are application-dependent and allow for significant flexibility in how the framework is used. For example, this can be used to allow Gyrus to work with offline (i.e., delayed) email

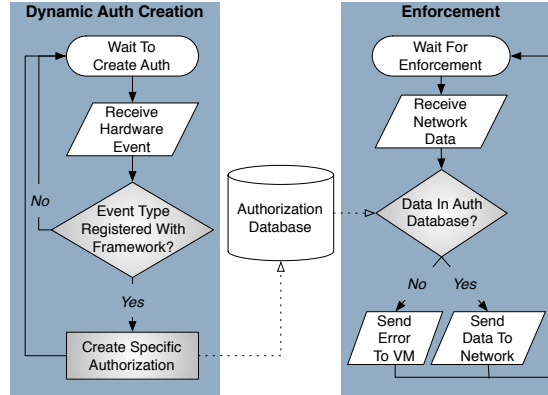


Figure 19: A high-level view of the Gyrus framework logic. Gyrus can be extended to support new applications through modules that specify logic for the three steps highlighted in dark gray. The remaining logic is provided by the framework.

sending.

5.4.3 Enforcement

After the authorization is in the database, the initial hardware event is sent to the User VM. At this point, the application that receives the event will generate and send some network traffic. This traffic is redirected to a transparent proxy in the Security VM. The proxy performs content analysis on the outgoing network traffic to determine if it is authorized to leave the system. This analysis is the enforcement complement to the authorization described above. Therefore, the Gyrus framework needs an *enforcement module* that performs this content analysis and makes a decision based on the analysis for each application that it supports.

5.5 Gyrus Implementation

We implemented a prototype of the Gyrus framework using the Xen hypervisor [16] and XenAccess [127] for introspection. The Security VM was running Debian Linux 5.0 and the User VM was running Windows XP Service Pack 2. The prototype was implemented using a combination of Python and Java, except where noted below. Figure 19 shows the high-level logic for the framework. There are two event-driven loops, one for authorization

creation and one for enforcement.

Gyrus can be extended to support new applications through modules that specify logic for the steps in Figure 19 that are highlighted in dark gray. We refer to the software that determines if an event requires processing as the *event testing module*. Similarly, the software that creates the authorization is the *authorization creation module*. And the software that makes the enforcement decision is the *enforcement module*.

5.5.1 Hardware Event Interposition

In a default Xen installation, all hardware events pass through a special control VM known as Domain 0, before being sent to a User VM. Within Domain 0, Xen creates an abstract model of the hardware devices to present each User VM with a common hardware interface. This model, known as the device model, is implemented in Xen using a modified version of Qemu [20]. Because the Gyrus framework needs access to hardware events, we chose Domain 0 as our Security VM. Therefore, the framework simply taps into the hardware events as they pass through the device model.

We inserted code in two places in Qemu, which is written in C, to extract keyboard and mouse events. At each of these locations, the code first opens a connection to a FIFO server (described below). Once the connection is established, the keyboard or mouse information is serialized and sent to the FIFO server. The information sent includes the key or mouse button associated with the event, whether the button was pressed or released, and the screen coordinates associated with the event (only for mouse events). In addition, we perform a screen capture of the User VM using the `vga_hw_screen_dump` function in Qemu. The location of this screen capture file is sent along with the other event information. The screen capture is made available to Gyrus modules and can be used to verify user intent, as described in Section 6.2. After sending this information, Qemu waits for a reply before passing the event on to the User VM.

The FIFO server is another process that is receiving these hardware events through the

interprocess communication (IPC) channel. In this case, the IPC is performed through a UNIX named pipe. The process receiving these events is the one responsible for invoking the dynamic authorization creation code. This process is described in the next section.

Beyond keyboard and mouse events, our prototype of the Gyrus framework can also use network events to trigger dynamic authorization creation. The default setting in Xen is for all network traffic to pass through a virtual network bridge in Domain 0. Using standard passive network inspection tools, we can view all network traffic to and from the User VM. This information is then passed into the same process that receives the keyboard and mouse events.

5.5.2 Dynamic Authorization Creation

The dynamic authorization creation step begins when the framework receives a hardware event, as described above. The first decision to make at this point is whether the event should be processed. Modules register a handler that calls the *event testing module* to determine if the module is interested in processing an incoming hardware event. Making this determination for keyboard events only requires checking the specific key(s) that was pressed and knowing which application in the User VM will receive the key event. Network events are more complex, but many tools exist for rebuilding network frames and searching for specific information within this traffic. The most complex scenario in our prototype is mouse events. Typically, one must associate the mouse button pressed and its coordinates with a specific application and UI widget where the mouse click happened. Gyrus provides support for modules to make these determinations.

Interpreting both keyboard and mouse events requires knowing which application in the User VM will receive these events. Using VMI, memory analysis, and knowledge of the Windows user interface implementation, we created software that can reconstruct the widgets and windows that are present on the screen in the User VM as shown in Figure 20. This reconstruction allows the framework, running in the Security VM, to know critical

pieces of information about a user’s interaction with her system.

By combining information from the Wine project [38] with our own reverse engineering of the Windows XP graphical subsystem, `win32k`, we developed a robust library, the *window mapper*, that uses memory analysis to determine the placement, size, and stacking order of graphical widgets on the screen. In Windows, the data structures representing widgets form a tree: each window has pointers to its next sibling and its first child, as well as a rectangle giving its top-left and bottom-right coordinates. The order of the sibling widgets determines the z-order; if a window or widget is “above” one of its siblings, it will appear earlier in the sibling list. Using this information allows the framework to know which window – and which application – will receive a given mouse event. We can also determine the specific user interface widget that is associated with a mouse event. For example, using the mouse event coordinates, the framework can determine if a particular mouse event will click on the button used to send email or refresh a web page.

For keyboard events, we determined that `win32k` stores information about the window currently in focus by storing a pointer to the currently active window in the data structure that represents the user’s desktop. From there, we can determine which specific widget inside a window is currently receiving keyboard input by examining its input queue. This capability allows Gyrus to determine precisely where the window manager will send a given keystroke. For example, this allows us to determine whether the user pressed ENTER on the address bar or the search bar of a web browser.

For the purposes of our prototype implementation, we obtained our knowledge of the Windows user interface implementation using reverse engineering. Although the difficulty of reverse engineering poses a challenge to the implementation and deployment of Gyrus, this problem is orthogonal to our current work. In Chapter 6, we discuss existing work and future research directions that may decrease the difficulty of obtaining such application specific knowledge.

After determining that Gyrus is interested in a particular hardware event, the framework

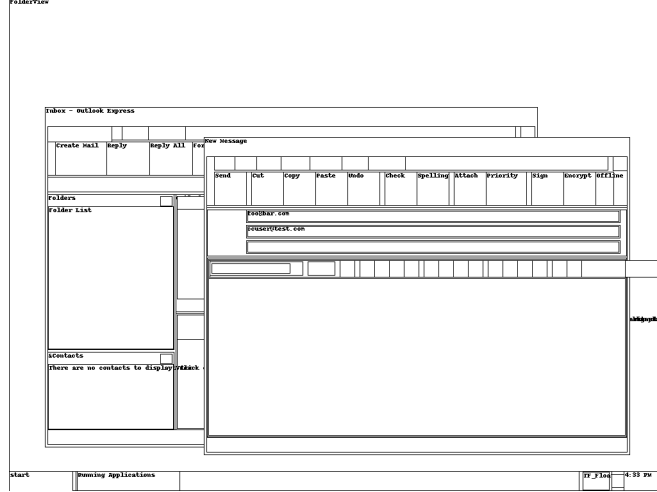


Figure 20: Windows user interface widgets as reconstructed in the Security VM to assist in mapping a given user input event to a specific application event handler.

will invoke the *authorization creation module*. We discuss this step in more detail in Chapter 6 because it is application-specific. After the authorization is created, Gyrus inserts it into a database and sends a reply to the device model, which passes the event to the User VM.

5.5.3 Enforcement

The Gyrus framework’s enforcement applies to specific network protocols. This is because each authorization in the database allows a specific piece of network traffic to leave the User VM. To provide enforcement, the framework must interpose on all network traffic leaving the User VM that is one of these specific network protocols. In our prototype, we redirect traffic based on port numbers. For example, email sending is performed using the SMTP protocol, which typically operates on port 25. Therefore, if a Gyrus module needs to interpose on SMTP traffic, then we redirect all traffic on port 25. While more robust techniques exist for protocol identification [55], they are beyond the scope of this thesis.

In Xen, as described above, all network traffic from the User VM passes through a virtual network bridge in the Security VM. Using the `iptables` tools in Linux, the framework transparently redirects the network traffic as needed based on the destination port number.

This traffic is redirected to a local transparent proxy that can interpret the data and make an enforcement decision. Each *enforcement module* must provide its own transparent proxy and the logic to approve or deny all outgoing network traffic seen by the transparent proxy. As seen in the case studies in Chapter 6, the transparent proxy can be an off-the-shelf product that serves this purpose (e.g., Squid [181] has a transparent proxy mode for HTTP traffic). Therefore, *enforcement modules* usually only need to implement the actual filter or module that performs the enforcement.

5.6 Summary

There are a variety of opportunities to extend or enhance the Gyrus framework. In this chapter we focused on a virtualization-based design and implementation for Gyrus, but it would be interesting to implement Gyrus on a co-processor or TPM-based platform with the goal of reducing the TCB size. In any of these cases, the overall goals would remain the same. Gyrus is designed to be a general purpose framework that enables security policy to provide a direct binding between a user's interaction with the system and subsequent hardware events. These hardware events can be network traffic (e.g., to stop a spam-bot), smart card operations (e.g., to stop malware from using credentials on a smart card), or anything else that a security application developer desires. By leveraging this binding, security policy no longer needs to model the entire system and can instead focus on identifying a single transformation that relates user-level information with hardware-level events.

The next chapter explores two detailed case studies of specific security applications that we designed and built using Gyrus. After describing the application case studies, we provide a security and performance evaluation for each to illustrate the overall viability of the Gyrus framework.

CHAPTER VI

APPLICATION CASE STUDIES USING GYRUS

6.1 *Motivation*

The Gyrus framework is flexible enough to handle any applications running in the User VM that receive user input and produce network output. However, supporting an application requires sufficient knowledge of the application logic to build the three key modules: *event testing*, *authorization creation*, and *enforcement*. For real-world deployment, we envision Gyrus and these modules being developed and distributed by a third-party organization such as a security vendor, similar to how anti-virus signatures are handled today.

For the purposes of testing Gyrus, we built support for two applications. Our primary goal in selecting these applications was to demonstrate the feasibility of supporting various types of applications and network protocols. In addition, we felt it was important to cover the applications and features that users are most likely to use on a regular basis. To make this determination, we analyzed data from the “Daily Internet Activities, 2000–2009” study by the Pew Internet & American Life Project [133].

The Pew Internet study asks people in the United States what Internet-related activities they performed on the previous day. Not surprisingly, nearly all of the activities were performed through an email client or a web browser. These activities included sending email, using a search engine, reading the news, checking the weather, online banking, watching videos, and more. Some activities mentioned in the survey that involve other client software include podcasting, instant messaging, phone calls, and peer-to-peer file sharing.

Given this usage data, we decided to build support for an email client and a web browser as these two classes of applications are clearly the most widely used. In addition, we

performed a study of how to support Internet-telephony applications. The remainder of this chapter describes the technical details of our support for each application and an evaluation of Gyrus' security and performance for each case study.

6.2 Application Case Studies

6.2.1 Email Case Study: Outlook Express

In order to demonstrate email application support, we extended Gyrus to support Outlook Express on Windows XP. Our modules for Outlook Express detect when a user is interacting with the application to send a message, and then extracts the message contents from memory and places it into a database of allowed messages. When an email is seen leaving the host, a transparent SMTP proxy checks that a message with matching content can be found in the database of authorized messages. This allows user email to pass unhindered, while blocking spam sent by malware on the host.

The implementation is divided into several components. First, the *event testing module* gets notification of hardware events, and decides if they represent a user sending an email. If so, the message's contents are extracted and then validated using the screen capture, and the *authorization creation module* creates an authorization allowing the message. Finally, in the *enforcement module*, the outgoing message is extracted from the SMTP session by the proxy, and the subject, sender, and body are compared with the authorization database.

6.2.1.1 Event Testing

In the event handler, the spam blocker receives notification of all mouse clicks from the device model, as described in Chapter 5. Upon receiving a mouse click event, the window mapper is consulted to see if the user is clicking on the "Send" button of an Outlook Express message window. Both a "left button down" and "left button up" event on the send button are required, with no intervening mouse button events. If the user is clicking on the send button, the system moves on to extracting the message contents.

6.2.1.2 Authorization Creation

To create a message-specific authorization, the message content is retrieved from both memory and the screen capture. Using memory analysis, we traverse the internal data structures used to represent a message while it is being composed. By reverse engineering portions of Outlook Express, we determined that the message composition pane is actually an instance of the MSHTML rendering engine (called Trident), which is also used by Internet Explorer to render web pages. When a user enters text into the window, the MSHTML engine dynamically updates the parsed HTML tree in memory with the new text. When the message is sent, the rendering engine serializes this tree to HTML and sends it using the SMTP protocol.

The parsed HTML is stored in memory as a splay tree [159], which optimizes access to recently used nodes. The nodes of this tree are objects of type `CTreePos`, and each tree node represents an opening or closing HTML tag or a text string (for the textual content of the page markup). HTML tags are represented by `CElement` objects (which are accessible from the corresponding `CTreePos`), which store, among other things, the name of the tag and its HTML attributes. Text nodes have no associated `CElement`, and are represented by their length and pointer into a document-wide *gap buffer*, a data structure commonly used to optimize interactive edits to a buffer. Our memory analysis code replicates the serialization process by traversing the tree and writing out the opening and closing tags, as well as the content of any text nodes. The same approach can be used to extract plain text email (by ignoring the HTML tags); however, we currently only implement the default case of HTML email. We also use memory analysis to retrieve the subject and recipients from the email client's "To" and "Subject" text boxes.

Since an attacker can manipulate the message contents in memory, as depicted in Figure 21, we validate the memory contents using their on-screen appearance. To do this, we use the bounding boxes of the subject, recipient, and message text from the window mapper to crop the screen capture provided by Gyrus down to just the text we are interested in.

Next, after upscaling and resampling the images to improve readability, we extract the text using the off-the-shelf Tesseract OCR software [160]. OCR is not completely accurate, however, so we use the Levenshtein edit distance [107] to compare the OCRed text with that retrieved from memory.

If the edit distance between the on-screen and in-memory strings exceeds a configurable threshold, the message validation fails and the message will not be placed into the authorization database. If the rate of OCR errors is sufficiently high, this could cause legitimate email to be rejected. In practice, we have found that setting an error threshold of 20% (relative to the length of the string) is sufficient to compensate for Tesseract’s mistakes. Note that this does not create much, if any, of an opportunity for an attacker to send spam because any alteration to the email will count against the 20% *in addition* to the OCR errors. Since the OCR errors approach 20% before any malicious changes to the email, an attacker would be unable to make any meaningful changes to the email.

If greater accuracy is needed, we could turn to the work of Lasko et al. [105] and use a Bayesian measure to estimate the probability that discrepancies between the strings are a result of OCR errors. We also note that Tesseract is not optimized for use with screen captures. Using an OCR algorithm that takes advantage of the fact that the image is computer generated could improve accuracy (e.g., if the font is known, even simple image matching might suffice to recognize individual characters). Once the message content from memory is validated, it is placed into the authorization database, and the mouse click is passed to the User VM.

Our current implementation uses a variety of costly operations (i.e., memory analysis, screen capture, OCR, etc.) whenever a user clicks the send button. In Section 6.4, we show that the combined time to complete these operations is approximately 2 seconds, and we discuss a variety of options for improving these numbers for greater user acceptability.

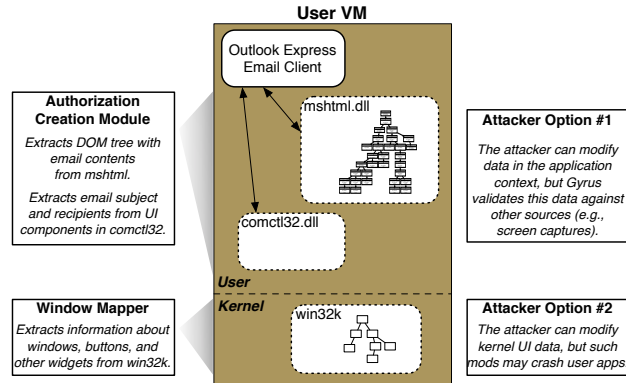


Figure 21: Gyrus' Outlook Express modules use VMI to extract data from the User VM memory. This data is interpreted and used to create an authorization for the user's email message.

6.2.1.3 Enforcement

At some point later (possibly much later, if the message is composed and sent while the user is offline), the message will be sent via SMTP to a mail server. When this occurs, an `iptables` rule on the virtual network bridge redirects the network stream to the transparent SMTP proxy (we use `proxsmtp` [176]), which calls our enforcement script. The script parses the message according to RFCs 2822 [137] and 2045 [66] and consults the authorization database to find messages with a matching subject and recipient.

Finally, the HTML part of the message is compared with the stored HTML from the database by recursively comparing each HTML node's content. The comparison between text nodes can be done exactly, because the copy in the database is extracted from memory and is not subject to OCR errors. Any message not found in the database is rejected with an SMTP reject (SMTP code 554: Transaction Failed). If the message is found and the contents match, it is allowed to be sent to the remote mail server. By placing authorizations in the database, we allow enforcement to occur at a time later than when the user sends the email, allowing for offline sending which improves the user experience.

6.2.1.4 *Discussion*

By detecting when the user has clicked on the send button and validating the message content against what the user sees on screen, we have confidence that we captured the user's intent. It is reasonable to assume that the message on screen when the user clicks "Send" is what he intended to send. As these are the only emails we allow the host to send, all spam is blocked from leaving the User VM when Gyrus is running. It is possible for some of the email text to be scrolled off the screen, in which case we are unable to completely verify the message contents in memory. We discuss the security implications of this situation in Section 6.3.

This general procedure can also be applied to web-based email. Using knowledge of the browser and webmail application semantics, we would use memory analysis to determine when the user clicks on the send button in the webmail client's composition page. As with a standalone email client, VMI could be used to extract the message text, validate it using the on-screen display, and place it in the authorization database. When the message is sent, an HTTP (rather than SMTP) proxy would be used to filter outgoing webmail messages to ensure they were generated by a human.

6.2.2 Web Browser Case Study: Internet Explorer

We also extended Gyrus to support Internet Explorer on Windows XP. These modules prevent malware generated click fraud by filtering outgoing HTTP traffic. This is a much more difficult problem than stopping spam, because the web browser provides a richer and more varied UI (with many UI actions that could result in an HTTP request), and because web content is more complex than email. Due to this added complexity, we currently only monitor a subset of UI actions that can be used to open a URL: hitting ENTER while the address bar has focus and clicking on a link in an open web page. However, the window mapper provides enough information to extend the range of monitored UI events to include other actions, such as clicking the "refresh" or "home" buttons, or selecting a bookmarked URL.

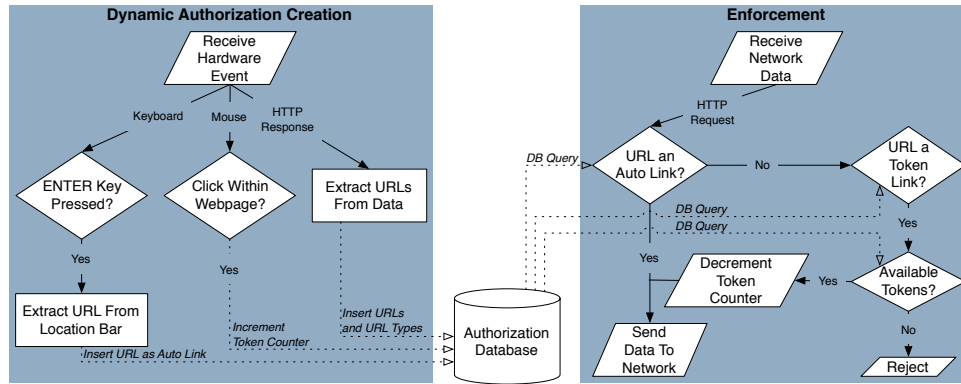


Figure 22: Logic followed by the *authorization creation module* and the *enforcement module* for web browser support within the Gyrus framework. Transitions not shown return to the starting state; these were omitted for clarity.

6.2.2.1 Event Testing

Clicking on the window and pressing ENTER on the address bar are handled by the *event testing module*. Upon receiving notification of the ENTER key being pressed from Qemu, the module uses the window mapper to see if the IE address bar is in focus. Likewise, when the mouse is clicked, we can use the window mapper to decide if the click occurs inside the IE content area, and ensure that no other window is covering the area the user clicked on. As with the send button, we require that both mouse-down and mouse-up events occur within the IE window, with no intervening events.

6.2.2.2 Authorization Creation

The case where a user types a URL into the address bar and hits ENTER is handled in much the same way as with our support for Outlook Express. If the event handler determines that the ENTER key was pressed when the address bar was in focus, we extract the URL and add it to the authorization database. As with the Outlook Express *authorization creation module*, we can also validate the URL from memory using the screen capture. The *enforcement module* can then check the outgoing HTTP request to ensure that it exists in the database.

Handling the case where a user clicks on a link in a web page is more difficult. Because web browsers show a rendered version of the underlying HTML, the visual representation

of a link may have nothing in common with the request generated by clicking on it. VMI is not useful in this case because there is no binding between the visual representation of the link on the screen and its representation in memory. Therefore, an attacker could alter the link target in memory, turning any legitimate user click into a fraudulent request.

To solve this problem, we turn to another source of input data: the incoming network stream. Like keyboard and mouse events, this input data is not under the control of an on-host attacker, and can be considered a hardware input event in the context of our framework. Incoming HTTP responses are parsed and their HTML content is analyzed using the JSoup [81] and EnvJS [136] libraries to extract URLs found in the returned web page. These URLs are divided into two categories: *automatic* URLs, which represent web page dependencies that will be automatically requested by the web browser, without any user interaction (for example, images and stylesheets), and *token* URLs, which will only result in an HTTP request if the user clicks on them.

Once the page links are categorized, the *authorization creation module* pre-approves all automatic links by adding them to the authorization database. This allows the web page to load normally for the user; all web page dependencies will be approved when the initial HTTP response is read, so the *enforcement module* will allow the traffic to pass as the browser makes additional requests to complete the page rendering. Figure 22 shows a flow chart depicting how the authorization and enforcement steps are linked by the authorization database.

Mouse clicks are then handled in the following way. The window mapper first checks to see if the click is within the IE page content widget. If so, the *token counter* is incremented in the authorization database, and the click is passed on to the guest OS. When the *enforcement module* sees an outgoing HTTP request, it checks to see if the requested URL is in the token URL database, and if there are any tokens available. If so, the request is allowed to pass, and the token counter is decremented. This ensures that every outgoing HTTP request is matched with a click on the web page. To further improve accuracy, we

could also make use of the information provided by the status bar to disregard clicks that are not on page links: when the user is not hovering over a link, the status bar will be empty, and this invariant can be used to detect clicks that will not generate a network request.

To ensure a strong binding between the user's interactions and the HTTP requests that we permit to leave the machine, we track the originating web page for each link in the database. When a new link is added to the database, we always note which page that link came from. This extra information is used in the *enforcement module*, to further limit an attacker.

6.2.2.3 *Enforcement*

The enforcement is performed using Squid [181] as a transparent HTTP proxy. Outgoing traffic from the User VM is redirected through the proxy using an `iptables` rule. Squid then uses the ICAP protocol [58] to obtain a decision from our *enforcement module*, which is implemented as a module to the Greasyspoon ICAP Server [116]. The *enforcement module*, as described above and shown in Figure 22, allows a request to go through only if a) the URL is in the automatic link database (i.e., it is a dependency of a previously authorized page), or b) the URL is in the token link database and there are tokens remaining. The *authorization creation module* treats requests that come from addresses typed into the location bar as automatic links; they are allowed even if no mouse click has occurred.

Thus, the first HTTP request made in the web browser will be authorized because the user must enter it using the address bar. Subsequent requests the user makes (as well as those made by automatic page dependencies) will be approved because the *authorization creation module* will have added the links as token (or automatic) URLs when the previous response was received. Each request is made either by clicking a link (which increments the token counter, allowing one request per click to go through) or by entering a URL into the address bar, so all legitimate attempts to visit a page will be allowed by Gyrus.

The *enforcement module* uses the originating web page for each link to ensure that

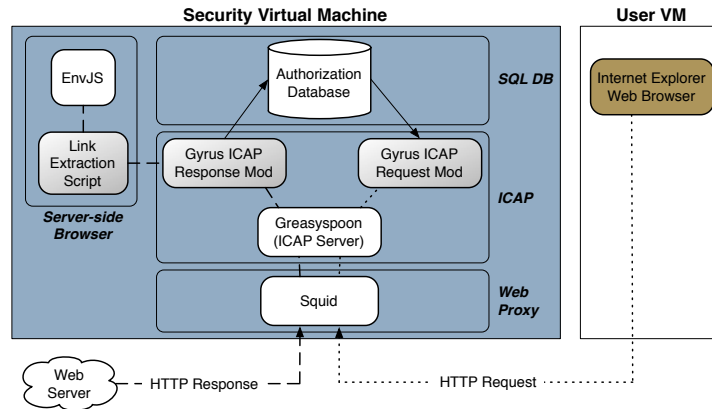


Figure 23: Software components involved in processing HTTP traffic with Gyrus. The four major software components (web proxy, ICAP server, SQL database, and server-side browser) are all easily interchangeable in the event that a better implementation becomes available. The shaded boxes indicate code written specifically for Gyrus as extensions to off-the-self open source software components.

the HTTP requests permitted at any given time correspond to links on the web page the user is currently viewing. Using the techniques described above, the *enforcement module* obtains the URL from the address bar at the time the HTTP request is being processed. This URL can, once again, be verified using the screen capture from Gyrus to protect against the malicious modification of memory in the User VM. Using this URL, the *enforcement module* will only allow the HTTP request if its originating web page matches the URL. Given that the URL in the address bar indicates the web page a user is currently viewing, this technique restricts the permitted HTTP requests to that page.

6.2.2.4 Discussion

Certain kinds of HTTP traffic pose special problems for our system. In particular, sites that use SSL to secure their transactions can not be monitored by a proxy without the encryption key. Our current implementation simply allows such traffic to pass unimpeded; a more sophisticated implementation could extract the session encryption keys from the browser's memory in order to examine these encrypted network streams.

An additional complication is introduced by the highly dynamic nature of many modern web pages. Such pages may use Javascript or other web scripting languages to modify the

content of the page after it is rendered; in some cases, dynamic code running on the page will even make its own HTTP requests (via XMLHttpRequest). Borders and Prakash also consider this problem in their study quantifying information leaks in outbound web traffic [24]. In order to predict the HTTP requests that will be made by a user's web browser, they used the Spidermonkey engine [167] to evaluate Javascript in incoming HTTP responses and extract any links from the resulting page. Although in the general case predicting links computed by Javascript is undecidable, their study was able to achieve coverage of 98.5% of traffic in practice. If complete Javascript coverage is needed, one could (at the expense of performance) render the web page in another VM, and automate clicks on all of the links, in order to extract all of the legitimate URLs [118].

In an effort to address the Javascript problem, we integrated a server-side web browser into the *authorization creation module* as shown in Figure 23. This browser, called EnvJS [136], is effectively the Rhino Javascript engine [166] integrated with a custom-built DOM, which is similar to the approach used by Borders and Prakash. The Gyrus ICAP response module, which handles all HTTP Response traffic, sends web pages to EnvJS so that they can be evaluated with Javascript before parsing the DOM tree for links. Because EnvJS is a relatively new project, we had limited success with this approach. Web sites, such as `http://www.facebook.com`, that extensively use Javascript did not properly load in EnvJS, resulting in a large number of false positives in our testing. We discuss the results of our testing with this software architecture in Section 6.3.

Given the modular design of our software architecture, there are two options for improving the system's performance in the future. First, one could wait for EnvJS to mature and become more robust at processing a wide variety of web pages. Second, one could replace EnvJS with another browser. This second option is potentially appealing because one could use a complete browser such as Mozilla Firefox or Google Chrome that would include support for not only Javascript, but also Adobe Flash, Java, etc. However, this functionality brings some extra costs as well. First, the performance of a complete browser

implementation would be slower than a lightweight system such as EnvJS. Second, as the browser environment in the Security VM grows larger, the security guarantees are reduced. Browsers such as Mozilla Firefox and Google Chrome are significantly larger than EnvJS and, therefore, likely have many more vulnerabilities. Ultimately, the choice comes down to a trade-off between performance, security, and functionality. The correct choice will likely vary based on the setting in which Gyrus is deployed.

6.2.3 VoIP Case Study: Skype

Although email and web browsing still account for most common desktop computing usage, voice over IP services such as Skype have grown to nearly 400 million users worldwide [57]. VoIP also represents an interesting test case for Gyrus: VoIP applications' interactions with the network are more complex than email or web browsing, and the binding between user input and network traffic is not as immediately clear. In addition, one of the most popular VoIP clients, Skype, makes use of a peer-to-peer overlay network to perform many of its functions, which increases the variety of traffic Gyrus needs to monitor and make decisions on.

For this thesis, we have not implemented a module to block malicious Skype traffic (such as VoIP spam, also known as SPIT). Instead, we will briefly provide here a sketch of what such an implementation would look like. Although the Skype protocol is officially undocumented, details of its workings have been uncovered through reverse engineering [49] and black box network analysis [17]. We base our analysis on the descriptions of the Skype protocol provided by these sources. Note that although Skype also implements an instant messaging service, we have chosen to omit it here for the sake of brevity; its operation is conceptually similar to the SMTP case, provided the user has already logged in.

Following the model used by Baset et al. [17], we divide Skype traffic into several categories: *login*, *user search*, *call initiation/teardown*, *media transfer*, and *presence messages*.

We will also adopt their notation for the various actors in the Skype network, using SN for a Skype Super Node, HC for the Host Cache, and SC for the Skype Client. In this section, we will describe how Gyrus might handle each type of traffic.

An initial hurdle is the pervasive use of encryption to protect the contents of Skype protocol messages. In order to successfully act on different messages sent by the Skype network, Gyrus would have to be able to decrypt both outgoing and incoming messages. In many cases, this task is not too difficult: although Skype uses RC4 to encrypt its UDP signaling packets, the key is derived from information present in the packet (specifically, the CRC32 of the source and destination IPs and a per-packet initialization vector), making it possible to de-obfuscate such packets without any additional data. For TCP packets, peers negotiate a longer-lived session key. This key is stored in the memory of the SC running inside the User VM, and can be recovered using VMI. These techniques would allow Gyrus to observe the decrypted contents of Skype traffic. We assume that this information is available to Gyrus for the remainder of the discussion.

When the Skype client starts, it attempts to make a TCP connection to a super node in order to join the P2P network. Connections are attempted to each of the SNs listed in the HC, which is stored in a file on disk; if the host cache is missing, the SC will fall back to a list of SNs embedded in the client binary. Once a connection to the overlay network is made, the SC then contacts the Skype login servers (which are centralized and hardcoded into the client) to perform user authentication. To support this phase of the protocol, Gyrus would need to whitelist the login and connection establishment messages sent to the login servers and the SNs in the host cache. The list of hosts to whitelist can be derived using VMI, so this phase is supported by the current framework.

Call establishment, teardown, and user search are a good fit for the current Gyrus framework. Call establishment is typically done by clicking the “Call” button while a contact is selected. Our existing framework is sufficient to monitor mouse clicks and detect when

they correspond to a click on the send button. Memory analysis can then be used to extract the username of the contact to whom the call is placed. The contact name can also be authenticated by comparing it against a screen capture. The network messages sent to initiate a call consist of an outgoing connection, either directly to the recipient or to an intermediate relay node. In either case, Gyrus could inspect the packet metadata to determine the eventual recipient of the call, and verify that it matches the name stored when the user clicked the call button. Call teardown and user search work in a similar way, and could be handled by Gyrus.

Once a call is established, the *media transfer* phase of the protocol begins, in which audio and (optionally) video data is transmitted to the call recipient. Due to the low-latency requirements imposed by real-time conversation, it is unlikely that Gyrus would be able to perform content analysis to verify that all the user's voice or video data had been faithfully passed on from the microphone or camera and onto the network. Instead, a Skype module would likely have to employ heuristics that estimated an upper bound on the outgoing traffic rate, based on input from the microphone and camera and knowledge of the codecs in use. However, this would still leave some window for an attacker to replace user content with his own while a legitimate call is in progress. To counter this threat, the Gyrus module should periodically sample a portion of the input and resulting network traffic, and compare them using an audio similarity measure offline. If a discrepancy is detected, Gyrus could terminate the call.

Finally, Skype periodically sends incidental network status updates, such as contact presence notifications and network keepalives, in order to maintain a connection to the Skype network. As these messages are not particularly useful to an attacker who wishes to send voice or video spam, we believe they can be safely whitelisted. With these measures in place, Gyrus should be able to effectively prevent Skype-based spam from being sent.

6.3 Security Evaluation And Analysis

New security frameworks should be secure against both current and future attacks. Here, we consider both scenarios for Gyrus by running current malware samples and analyzing the framework's security properties. We also discuss Gyrus' false positive and false negative rates using our prototype email client and web browser support.

6.3.1 Current Attacks

While running Gyrus with support for Outlook Express, we infected the User VM with `Spammer:Win32/Cutwail.gen!B`, a spam bot. We monitored all network traffic to and from the User VM to ensure that nothing escaped our analysis. Shortly after infection, the spam bot started sending out a large volume of spam messages. However, Gyrus successfully stopped all of these messages. While the spam bot was attempting to send its messages, we also composed a legitimate email and successfully sent it. Gyrus approved this message and allowed it to be delivered to the intended recipient. These results are intuitive because the spam bot can not create hardware events to authorize its messages. Since other spam bots would operate in the same manner as this one (i.e., sending spam to remote email servers using SMTP but unable to produce hardware events), they would also be stopped by Gyrus.

While running Gyrus with support for Internet Explorer, we infected the User VM with a variety of click bots (one at a time) including `AdClicker-AD, DR/Click.HSP.A.2`, and `AdClicker-BY`. Many of the bots failed immediately because they needed to obtain command and control information using HTTP, which was blocked by Gyrus since these requests were not initiated by the user. However, the three bots listed above utilized hard-coded click fraud targets. Gyrus also stopped this traffic. While the bot traffic was being blocked, we were still able to browse web pages normally using Internet Explorer. As with the spam testing, these results are intuitive because the click fraud bots can not create hardware events to authorize their HTTP requests.

Both of these bots produced network traffic that is similar to human activity. However, using the Gyrus framework, we stopped all of the malicious traffic while allowing all of the legitimate traffic to leave the host.

6.3.2 Future Attacks

In order to understand the Gyrus framework's susceptibility to future attacks, we return to the original framework requirements, threat model, and assumptions discussed in Section 5.3. We start by looking at the initial hardware events that result in the creation of new authorizations. Our threat model assumes that these events are non-forgable. However, attacks against the virtualization architecture could result in an attacker gaining access to the Security VM, which would allow an attacker to forge hardware events to Gyrus. Alternatively, an attacker could modify the firmware on the system's hardware devices to forge hardware events. The firmware attack is challenging and would have limited impact (requiring a new attack for each different hardware device). Therefore, the larger threat is attacks against the virtualization architecture. For this reason, we stress that the hypervisor is part of the TCB. Gyrus depends on the isolation provided by the hypervisor (as do all virtualization-based security frameworks), so it is critical to ensure that it is deployed on a hypervisor with a small attack surface area and a well audited code base. Even though modern hypervisors have significantly fewer discovered vulnerabilities than full featured operating systems, as additional features creep into these hypervisors, it may be prudent to develop a custom hypervisor for use in security frameworks to provide higher assurance to the TCB.

The next avenue for attack is manipulation of the authorization creation algorithms. Using knowledge of the information gathered to create authorizations, an attacker could attempt to get a malicious authorization placed into the database by manipulating data in the User VM memory. This attack is very limited in scope. Since Gyrus validates data from memory with other sources (e.g., network traffic or screen captures), the attacker

would need to manipulate data in multiple places, and a user is likely to notice any visible changes to his content. Although this attack could be made more stealthy by waiting until the user is about to send a message (i.e., by detecting when the mouse pointer is over the Send button in the mail client), over time the user is likely to notice something amiss and take corrective action by cleaning his system.

An attacker could also exploit the fact that long email messages can scroll off the screen. In such cases, the email contents in memory could be altered to append spam content to the user's message because the screen validation would not provide complete coverage. However, the attacker would still be restricted to the user's intended subject and recipients, and would still be rate-limited by the legitimate hardware events. Figure 21 shows the software components that an attacker would need to manipulate for this attack.

Another way for an attacker to manipulate the authorizations would be to change the layout of memory in the User VM. Changing the locations and format of data structures in memory would violate the introspection assumption, allowing an attacker to arbitrarily manipulate the memory data seen by Gyrus. However, even if an attacker successfully completed this very challenging attack, she would still need to modify the contents on the screen or the contents of incoming network data. As described above, this would likely lead to detection.

The final option for an attacker is to craft outgoing network traffic to bypass or pass the enforcement step. Given the specificity of the authorizations, the attacker cannot pass the enforcement step without actually being legitimate (e.g., sending an HTTP request to a web page that the user was going to visit anyway, or sending an email that the user was already planning to send). Bypassing the enforcement would require disguising the outgoing traffic in order to prevent protocol analysis. Encryption would be sufficient for this purpose, however this would preclude using the User VM to independently complete the malware's goals. Specifically, an intermediate host would have to decrypt the traffic before sending it to the final destination, reducing the User VM's value to the attacker.

In summary, attackers must execute multiple challenging attacks to defeat the Gyrus framework. This significantly improves system security compared to previous related research.

6.3.3 False Positive and False Negative Analysis

Gyrus is designed to allow all user-initiated network traffic originating from the applications it supports. Therefore, a false positive occurs when any legitimate user-initiated traffic is blocked. Running Gyrus with Outlook Express, we did not have any false positives. All legitimate emails were allowed. Running Gyrus with Internet Explorer, we logged every HTTP request and noted which ones were denied. For a single web page, there could be tens or hundreds of requests to load every aspect of the page (e.g., images, style sheets, javascript files, etc.). With our current implementation, we have a false positive rate of 81.9% when viewing the top 1000 most popular sites from Alexa. This is clearly too high for practical everyday usage. As discussed in Section 6.2.2, most of these false positives are a result of EnvJS incorrectly processing real world web pages. As EnvJS matures, we anticipate that this number will decrease. Surprisingly, even with this high of a false positive rate, in many cases web sites were still usable, only missing one or two images (blocked due to false positives). In other cases, the blocked requests resulted in the website not loading sufficiently to be usable.

After EnvJS matures there will still be similar problems related to supporting Adobe Flash, Java, conditional comments, and other obscure language features. In short, to get zero false positives, we would need the complete rendering capabilities of the user's web browser. We discussed the benefits and drawbacks of this approach in Section 6.2.2.

A false negative, in the context of Gyrus, is when malware is able to send network traffic that is approved by the system. Our testing, as described above, showed zero false negatives when we tested on existing bots. Malware designed with Gyrus in mind could communicate using covert channels. While Gyrus does not remove all covert channels

Table 3: Outlook Express latency introduced by Gyrus for different user interactions. All times are in milliseconds.

User Interaction	Mean	Std Dev
Click while OE not running	3.8	0.42
Click with no OE compose window	23.7	0.48
Click in compose edit area	28.0	0.00
Click in compose tool bar	109.9	0.32
Click on send button	2067.6	73.17

Table 4: Internet Explorer latency introduced by Gyrus for different user interactions. All times are in milliseconds.

User Interaction	Mean	Std Dev
ENTER while IE not running	2.0	0.00
ENTER while focus on search bar	2.5	0.53
ENTER while focus on location bar	456.7	10.37
Click while IE not running	3.8	0.42
Click in IE window (not web page)	28.1	0.32
Click on web page	35.3	0.95

from the host, it does significantly constrain them. All data sent from Gyrus-protected applications must conform to constraints imposed by both the user input and the application semantics. Furthermore, since all of Gyrus-protected traffic travels through a transparent proxy, timing-based covert channels are more challenging to implement. Therefore, with Gyrus, an attacker would be limited to minor protocol manipulations and other low bandwidth covert channels.

6.4 Performance Evaluation

We tested Gyrus on a dual quad-core Intel Xeon E5450 computer with 8GB of RAM. The system ran Xen 3.4.2 with Gyrus patches applied to the device model code (i.e., qemu). The User VM ran on one processor core with 384 MB of RAM. The Security VM used the remaining system resources.

Since the Gyrus framework interposes on hardware events, the key performance metric

that we are concerned with is the latency introduced for this added security. Tables 3 and 4 show the latency introduced for various user interactions with Gyrus running. Each Gyrus extension performs a different amount of processing based on the current system state and the hardware event that it is handling. For example, the smallest amount of latency is introduced when the corresponding user applications (e.g., Outlook Express or Internet Explorer) are not running. This is because the Gyrus extension can quickly determine that no further analysis is needed.

Latency times that are acceptable to users, as established by prior research, fall within the range of 50 – 150 ms [157]. Most of the latencies listed in Tables 3 and 4 fall within or below this established range of acceptability. However, the time required for processing when a user clicks on the send button in Outlook Express is just over 2 seconds. This is because of the substantial amount of work performed by the *authorization creation module* while handling this hardware event. As described in Section 6.2, this step involves a screen capture, significant memory analysis, OCR operations, edit distance computations, and database operations. Unfortunately, users are less concerned with these technical details, and would simply find this latency unacceptable.

Given this performance problem, we investigated options for reducing the latency as shown in Figure 24. The baseline in this figure shows the current latency, which is the same as that reported in Table 3. The other bars show the latency imposed by alternative options. The premise behind each alternative option was that the costly memory analysis and OCR operations could be handled in a separate thread after capturing the system state and sending the hardware event to the User VM. If we could capture the system state quickly, then the perceived latency from the user's perspective would be significantly reduced. This idea is especially viable on modern multi-core systems because the analysis could happen in parallel, effectively imposing little to no overhead to the user. This idea hinges, of course, on the ability to quickly capture the system state.

The system state that we need consists of a memory snapshot and a screen capture.

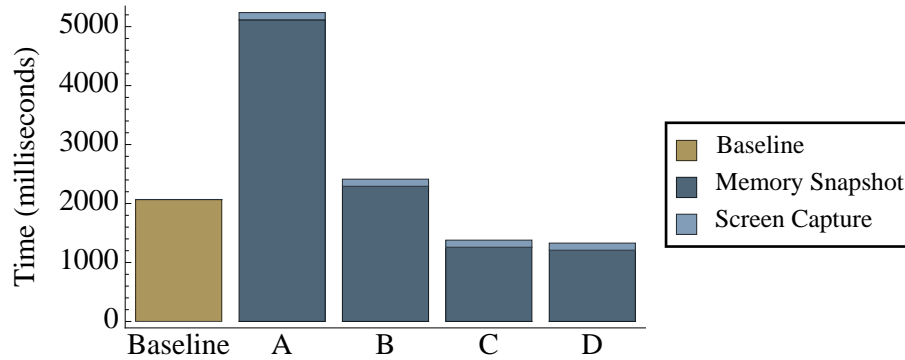


Figure 24: Time required for work performed by Gyrus when a user clicks on the send email button. The baseline shows the time for doing all analysis inline. The remaining bars show the time required to simply save the screen and memory information, allowing it to be processed in parallel using a separate thread. Option A uses `xm save -c` to perform a memory snapshot. Option B uses the `dump_memory.c` example program from XenAccess. Option C uses a modified version of `dump_memory.c` that stores the snapshot in memory instead of on disk. And option D parses the page tables on the target system to only store the pages needed for our analysis. Each option requires saving the screen buffer, which requires an additional 122ms and is shown at the top of each stacked bar.

Our screen capture algorithm is already optimized and is limited primarily by the speed of writing the image to disk. This operation takes 122ms, and could probably see a marginal speedup if we instead wrote the image to memory, but this savings would be dwarfed by the overhead of the memory snapshot. Therefore, we focused our efforts on improving the time required to obtain the memory snapshot. Figure 24 shows performance measurements for four options:

- **Option A:** Xen provides the `xm` tool chain as a management utility. One option for this tool allows you to save the current state of a virtual machine, including both the memory and CPU state. When run as `xm save -c <domid> <output file>`, this tool will simply perform the snapshot and save it to a local file. This operation took an average of 5116ms in our testing, which is slower than the baseline so it is not a viable option.
- **Option B:** XenAccess, which is part of the Turret architecture discussed in Chapter 3, includes an example tool that dumps the memory from a running virtual machine into

a local file. This example is called `dump_memory.c`. This operation took an average of 2292ms in our testing, which is slower than the baseline so it is not a viable option.

- **Option C:** We hypothesized that `dump_memory.c` could be made faster by saving the snapshot to local memory, instead of to a file. To test this theory, we modified the `dump_memory.c` example code to save the image to memory. This operation took an average of 1258ms in our testing. This is faster than the baseline, but still not fast enough to be acceptable to users.
- **Option D:** Next, we hypothesized that we could improve performance by only saving the portions of memory that are needed in our subsequent analysis. To test this idea, we implemented a reference program that identifies all of the physical memory pages for a given process, then saves those pages into local memory. The task of locating all of the physical memory pages was more costly than anticipated, resulting in an overall average time of 1208ms in our testing. Similar to Option C, this is faster than the baseline, but still not fast enough to be acceptable to users.

Our fastest memory snapshot technique is still two orders of magnitude slower than what we need for this application. We believe that the only way to achieve the desired performance is to use a copy-on-write memory snapshot technique. In theory this should provide the level of performance that we require. However, copy-on-write support in Xen is currently very immature and only available as an unmaintained, third-party patch [39]. We obtained this patch to test on our system. However, we were unable to achieve the desired functionality and the patch author has thus far been unable to identify the cause of the problem. Even so, we believe that copy-on-write memory snapshots are the correct solution to this problem. At this point, given the general usefulness of the copy-on-write snapshot technique, it is only a matter of time before it is incorporated into Xen in a more robust and stable fashion.

6.5 Discussion and Future Work

The Gyrus framework and our application support for Outlook Express and Internet Explorer all required a significant amount of software engineering. Given the inherent complexities of techniques such as transparently intercepting web traffic or converting a screen capture image into usable text with OCR, we utilized as much off-the-shelf and open source software as possible. Overall, this approach has worked well and allowed us to achieve our primary goals. However, based on our experiences with the system at this point, there are a few places where we believe that replacing an off-the-shelf component with a custom-built component could positively impact Gyrus's performance.

One such example is with the OCR algorithm. Most OCR algorithms, including the one we used with Gyrus, are designed for extracting text from a scanned document. Extracting text from a screen capture is a slightly different problem. And this difference accounts for the higher error rate that we are seeing through OCR. Replacing our algorithm with one designed for use with screen captures should reduce the error rate, allow for tighter tolerances in our edit distance computations, and ultimately increasing the overall security of the system.

Another example where custom-built software could help is with the VM memory snapshots. As described in Section 6.4, none of the standard memory snapshot techniques are fast enough for our application. Copy-on-write snapshots appear to be the best answer, but this feature is not currently included with Xen. Therefore, work on this component would not only benefit Gyrus, but also the greater Xen community.

Support for the various web technologies remains a difficult task. The current Gyrus software architecture is well poised to benefit from any third-party advances in this field. But this may also be an area where custom-built software and/or contribution to an open-source project could benefit both Gyrus and the open source community.

Looking beyond our current support for Outlook Express and Internet Explorer, one of the key challenges with Gyrus is the level of effort required to support new applications.

Currently, this requires expert reverse engineering to understand the application semantics. And this semantic understanding must be updated with each new software version. Techniques such as dynamic software analysis can help by semi-automating the task of locating key data structures in memory, but these are only likely to work with simpler classes of applications. In general, we can think of the user applications as falling into one of three classes:

1. **Class 1 – Easy:** Applications that can record keyboard input and network output and have a computer program explain the transformation (e.g., instant messaging).
2. **Class 2 – Medium:** Applications where transformations are more complex and require additional application knowledge (e.g., email clients).
3. **Class 3 – Hard:** Applications where network output depends on more than just user input and/or transformations are complex across both time and input boundaries (e.g., web browsers).

Using this classification, we believe that automated or semi-automated techniques for application support are possible for both Class 1 and Class 2 applications. For example, recent work by Jung and Clark [92] has produced methods to infer the data structures used by applications. This would allow us to automate much of the work that, for example, allowed us to understand that MSHTML used a splay tree for its storage of HTML content. In addition, work done on malware analysis [156] and protocol reverse engineering [30] has begun to produce automated techniques for inferring some higher level application semantics as well. These advances should make the challenge of reverse engineering more tractable and enable easier creation of new Gyrus modules.

The final deployability concern with Gyrus is related to its use of a virtualization-based architecture. Our current system is built on Xen which is difficult to install and somewhat limiting in its ability to work with desktop hardware and laptops. However, Xen is not required to run Gyrus. Instead, one could engineer a small virtualization layer that is similar

to Blue Pill [145] that can easily install under the user’s existing operating system. This technology exists, but would need to be adapted to work with Gyrus. The benefits of this approach would be greatly simplified Gyrus installation and, likely, improved performance throughout the system.

6.6 *Summary*

In the previous two chapters, we introduced the Gyrus framework and showed how it can be used to distinguish between human and malware generated network traffic for a variety of applications. By combining secure hardware monitoring with virtual machine introspection and memory analysis, we linked human input to observed network traffic and used this information to make security decisions. Using Gyrus, we demonstrated how to stop spam and click fraud attacks. Our evaluation demonstrated that Gyrus successfully stops modern malware, and our analysis shows that it would be very challenging for future attacks to defeat it. Finally, our performance analysis shows that Gyrus is a viable option for deployment on desktop computers with regular user interaction. Gyrus fills an important gap, enabling security policies that consider user intent in determining the legitimacy of network traffic.

CHAPTER VII

CONCLUSIONS

7.1 Summary and Contributions

In this thesis, and in the supporting research, we set out to reimagine security in modern systems. Previously, security has been an “all or nothing” proposition. One option was to design and build a high assurance system, which often served a small number of purposes, at a great expense. And the other option was to use commodity products with a minimal expectation of security. The work presented here falls somewhere in the middle. Our goal, simply stated, was to allow the use of commodity software while providing an improved security posture to the overall system. We achieved this goal through the use of external monitoring techniques, in which security software is protected from attack through a small, trusted layer of software known as a hypervisor.

Our Turret architecture builds on top of a virtualization-based architecture, allowing one virtual machine to run security tools and benefit from a controlled isolation from the rest of the software on the platform. Other virtual machines on the same platform run normally and simply benefit from improved monitoring and security checking. Turret enables this idea through a variety of mechanisms designed to facilitate secure active monitoring from within this protected vantage point.

The monitoring capabilities provided by Turret are comprehensive. From within the Security VM, we can view the entire memory space and any hardware events (e.g., network traffic, hard disk reads and writes, keyboard and mouse activity, screen buffer, etc.) from each User VM. In addition, we can install protected software hooks inside the User VM that provide event-driven notifications about activity in that system while ensuring that malicious software cannot disable or circumvent the hooks.

Overall, the Turret architecture has been very successful. Portions of this architecture were published in two conferences [125, 126]. The passive memory monitoring techniques are now available as a freely available open source library called XenAccess. And, most importantly, these techniques have gone on to be used in other research projects [65, 180], commercial products [173], forensic analysis tools [27], and a variety of other domains. The primary research contributions of the Turret architecture include:

- The XenAccess open source library, which is now the only virtual machine introspection solution available for Xen (Section 3.5.1).
- The ability to perform secure active monitoring by placing protected hooks within the User VM's operating system (Section 3.4).
- A series of design principles to guide the creation and use of external monitoring for security applications (Section 3.3.1).
- The integration of comprehensive and secure monitoring capabilities into a single system (Section 3.4).
- An extensive security analysis of the Turret architecture showing that it raises the bar for the security of commodity systems (Section 3.6).

One of the key drawbacks to external monitoring is the semantic gap problem. Without access to the programmer interfaces (APIs) within a system, it is much more difficult to extract the information needed for security monitoring. It is called a semantic gap because the information available is low-level (e.g., raw memory), but we desire high-level semantics (e.g., a list of the running processes) about the system. The semantic gap is a large problem that is beyond the scope of a single dissertation. In this thesis, we provided an overview of the current techniques used for memory analysis in Section 4.2. Then we described a new technique that uses machine learning to build models of data structures in memory.

Using these models, we can locate a data structure in memory across a variety of software versions. The primary contributions of our memory analysis work include:

- The use of hidden markov models to model data structures in memory (Section 4.3.2).
- An architecture for integration of machine learning techniques into runtime memory analysis and virtual machine introspection (Section 4.3).

The remainder of this thesis focused on using external monitoring for a specific class of security applications. Our application framework, called Gyrus, monitors human input events (e.g., keyboard and mouse activity) and uses this information to apply security decisions to outgoing hardware events (e.g., network traffic, smart card access, etc.). The Gyrus framework is general, allowing for support of new applications through creation of an *event testing module*, an *authorization creation module*, and an *enforcement module*. For this thesis, we implemented support for the Outlook Express email application and the Internet Explorer web browser by creating the three required modules for each application. Our support for each of these applications is designed to only permit outgoing network traffic when it can be directly correlated with human input events (e.g., a user clicking on the send button in an email client, or a user clicking on a link in a web browser). The primary contributions of the Gyrus framework include:

- A general framework for correlating non-forgable human input events with hardware events leaving a virtual machine (Section 5.4).
- Implementation and analysis of the software modules needed to extend the Gyrus framework to support Outlook Express and Internet Explorer (Section 6.2).
- A complete security and performance analysis of the Gyrus framework showing its benefits and limitations for real world deployments (Section 6.3 and 6.4).

7.2 *Open Problems*

As is the case with any research, this thesis both solves problems and introduces new problems. Furthermore, since we are rethinking how security applications should be deployed, there are a variety of highly practical questions regarding the usefulness, the long-term viability, and the benefits of the ideas presented in this thesis. Our research approach is to embrace these challenges as ideas for future work, rather than to view them as obstacles or even road blocks. Throughout this thesis we have described a variety of opportunities for future work. In this section, we take a high-level view of the research space related to external monitoring and identify the key open problems in this space. These problems are both large and challenging, but success could fundamentally change system security for the foreseeable future by allowing external monitoring techniques such as virtual machine introspection to be more readily deployed in commercial security products.

- **Solve the Semantic Gap Problem:** The semantic gap problem is currently the single largest obstacle to deploying VMI applications. In this thesis we investigated a piece of this problem using machine learning techniques, but we also acknowledge that this approach cannot achieve perfect accuracy and may not be suitable for all applications. There are a variety of possibilities for solving the semantic gap problem. One option could be to partner with software companies to get the appropriate semantic information about each program to be exposed or made publicly available. Another option could be to develop software analysis techniques that can automatically extract the necessary semantic information from source code or, ideally, binary versions of software. A third option could be to create a public semantic repository where reverse engineers can deposit discovered semantic information using a well defined data format. There are potentially many other options, and the final solution may be a hybrid of these approaches. However, without a unified solution to this problem, VMI will never achieve broad general acceptance.

- **Simplify VMI Programming:** VMI code is currently very messy. As an example, consider Listing 1. The problem is that the code is directly manipulating raw memory resulting in both the memory layout and the software logic being embedded throughout the program. This creates code that is difficult to read and difficult to maintain. This can be addressed using a programming language that specifically separates the code logic from the memory layout [187], or through a series of interfaces that progressively build higher semantic abstractions on top of the raw memory. The correct path forward for addressing this problem is not obvious. A new language could provide more flexibility, but may require porting a lot of legacy code. However, building new APIs can be a slow process because the end result is somewhat rigid and a successful API will need to account for a variety of competing interests.
- **Create a VMI-Specific Virtualization Platform:** All of the work in this thesis was performed on Xen, which is a server-class virtualization solution. This approach offered the benefit of being able to focus on VMI, without spending time building a new hypervisor. However, Xen may not be a practical long-term solution for VMI applications. First, Xen is too large to provide strong (i.e., verifiable) security guarantees. Next, since Xen is not designed for desktop systems, its performance is poor in these settings when things such as video processing are critical. And, finally, Xen is challenging to install on many platforms including laptops and desktop systems. A solution to all of these problems is to build a hypervisor that is designed specifically to provide controlled isolation to a Security VM and to enable the Turret architecture to operate within that VM, without a challenging installation procedure. Additional virtualization features could be made available by allowing layered hypervisors, if needed.
- **Create Audit-Aware Software:** In addition to solving the semantic gap problem, as external monitoring becomes more popular it will become increasingly important to

build software so that it can more easily be audited. Ideally, this should not be left to the application developers. Instead, this should become a new feature in compilers. A compiler could generate a program such that its layout in memory at runtime is optimized for external monitoring. This goal could be achieved by controlling where and how data is laid out on each page of memory and embedding semantic information alongside data in memory. Compilers could also output all of the necessary semantic information needed to solve the semantic gap problem. This information could be placed in memory or added to a database repository used by VMI software.

- **Enhance and Generalize VMI Libraries:** As a part of our research, we made the XenAccess library publicly available. However, this library only works on Xen and does not include any of our secure active monitoring technologies. VMWare provides its own VMI library, VMsafe [173], that only works on VMWare products and is incompatible with XenAccess. So currently application developers must chose which virtualization platform to work on at the start of their project, and changing that decision later is difficult and costly. Ideally, we believe that XenAccess should be expanded to include support for the secure active monitoring techniques discussed in this thesis. In addition, XenAccess should be ported to work on Xen and KVM, the two major open source virtualization platforms. And, finally, a common API should be implemented as a generic VMI library that works with both XenAccess and VMWare’s VMI library. These changes, taken together, will make VMI a commodity service that is available on all of the major virtualization platforms. Developers can then create one version of their VMI-enabled software and it would work on any of these platforms. While these steps do not represent major research challenges, they do represent important engineering challenges that will lead to the further acceptance and adoption of VMI.

7.3 Closing Remarks

This thesis explored the use of external monitoring to improve system security. We have presented Turret, an extensive monitoring framework designed with security applications in mind. Using Turret as a foundation, we explored both memory analysis techniques and novel security applications. Looking ahead, external monitoring appears to be a promising technique that may even change the way systems are designed and deployed in the future. The open problems discussed above are the first step toward moving the community down this path.

REFERENCES

- [1] “Bit9 Parity Application Whitelisting.” <http://www.bit9.com>. 5.2.2
- [2] “ZoneAlarm.” <http://www.zonealarm.com>. 5.2.2
- [3] ABADI, M., BUDI, M., and LIGATTI, Ú. E. J., “Control-Flow Integrity,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2005. 2.1.1
- [4] ALVES-FOSS, J., TAYLOR, C., and OMAN, P., “A multi-layered approach to security in high assurance systems,” in *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2004. 2.1.1
- [5] AMES, S. R., “Security kernels: A solution or a problem?,” in *Proceedings of the Symposium on Security and Privacy*, 1981. 2.1.1
- [6] ANDERSON, J. P., “Computer security technology planning study vol 1,” Tech. Rep. ESD-TR-73-51, Vol 1, ESD/AFSC, Bedford, MA, October 1972. 2.1.1
- [7] ANDERSON, J. P., “Computer security technology planning study vol 2,” Tech. Rep. ESD-TR-73-51, Vol 2, ESD/AFSC, Bedford, MA, October 1972. 2.1.1
- [8] ARASTEH, A. R. and DEBBABI, M., “Forensic memory analysis: From stack and code to execution history,” in *Proceedings of the Digital Forensic Research Workshop (DFRWS)*, August 2007. 2.4
- [9] ARBAUGH, W. A., FARBER, D. J., and SMITH, J. M., “A secure and reliable bootstrap architecture,” in *Proceedings of the 1997 IEEE Symposium on Computer Security and Privacy*, May 1997. 2.1.1, 3.3.3
- [10] ASRIGO, K., LITTY, L., and LIE, D., “Using VMM-based sensors to monitor honeypots,” in *Proceedings of the International Conference on Virtual Execution Environments (VEE '06)*, 2006. 2.2.2
- [11] AXELSSON, S., “Intrusion Detection Systems: A Survey and Taxonomy,” Tech. Rep. 99-15, Chalmers University of Technology, 2000. 5.2.2
- [12] BACIC, E. M., “The canadian trusted computer product evaluation criteria,” in *Proceedings of the Annual Computer Security Applications Conference*, 1990. 2.1.1
- [13] BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., and HAGHIGHAT, S. A., “Practical domain and type enforcement for UNIX,” in *Proceedings of the Symposium on Security and Privacy*, 1995. 2.1.1

- [14] BAIARDI, F. and SGANDURRA, D., “Building trustworthy intrusion detection through vm introspection,” in *Proceedings of the Third International Symposium on Information Assurance and Security (IAS)*, 2007. 2.2.2
- [15] BALIGA, A., GANAPATHY, V., and IFTODE, L., “Automatic inference and enforcement of kernel data structure invariants,” in *Proceedings of the Annual Computer Security Applications Conference*, 2008. 2.1.1
- [16] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARFIELD, A., “Xen and the art of virtualization,” in *Proceedings of the Symposium on Operating System Principles*, October 2003. 2.3, 3.2.2, 5.5
- [17] BASET, S. A. and SCHULZRINNE, H., “An analysis of the Skype peer-to-peer internet telephony protocol,” in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2006. 6.2.3
- [18] BAUM, L. E., PETRIE, T., SOULES, G., and WEISS, N., “A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains,” *The Annals of Mathematical Statistics*, vol. 41, pp. 164 – 171, February 1970. 4.3.2
- [19] BEDDOE, M., “The protocol informatics project,” in *Toorcon*, 2004. 4.2.4
- [20] BELLARD, F., “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005. 3.5.2, 5.5.1
- [21] BELLMAN, R. E., *Adaptive Control Processes*. Princeton University Press, 1961. 4.3.1
- [22] BELLOVIN, S. M., “Virtual machines, virtual security?,” *Communications of the ACM*, vol. 49, p. 104, October 2006. 2.3
- [23] BORDERS, K., *Protecting Confidential Digital Information From Malicious Software*. PhD thesis, University of Michigan, 2009. 5.2.2
- [24] BORDERS, K. and PRAKASH, A., “Quantifying information leaks in outbound web traffic,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009. 6.2.2.4
- [25] BORDERS, K., ZHAO, X., and PRAKASH, A., “Siren: Catching Evasive Malware (Short Paper),” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2006. 5.2.1
- [26] BOVET, D. P. and CESATI, M., *Understanding the Linux Kernel*. O’Reilly & Associates, Inc., 3rd ed., 2005. 3.5.1.3, 3.8.1
- [27] BRUECKNER, S., “Novel applications of xen: Virtual training & malware evaluation,” in *Xen Developer’s Summit*, June 2008. 7.1
- [28] BURDACH, M., “Digital forensics of the physical memory.” <http://forensic.seccure.net>, July 2005. 2.4, 4.2.2
- [29] BURDACH, M., “An introduction to windows memory forensic.” <http://forensic.seccure.net>, July 2005. 2.4, 4.2.2

- [30] CABALLERO, J., POOSANKAM, P., KREIBICH, C., and SONG, D., “Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering,” in *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2009. 6.5
- [31] CASAROSA, V. and PAOLI, C., “VHM: A Virtual Hardware Monitor,” in *Proceedings of the Workshop on Virtual Computer Systems*, 1973. 2.2.2
- [32] CHEN, K., “Reversing and exploiting an apple firmware update,” in *Proceedings of Black Hat USA*, 2009. 5.3.2
- [33] CHEN, P. M. and NOBLE, B. D., “When virtual is better than real,” in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOT OS)*, May 2001. 2.3
- [34] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., and PORTS, D. R. K., “Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008. 2.1.1
- [35] CHESWICK, W. R., BELLOVIN, S. M., and RUBIN, A. D., *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2nd ed., 2003. 5.2.2
- [36] CHRISTODORESCU, M., SAILER, R., SCHALES, D. L., SGANDURRA, D., and ZAMBONI, D., “Cloud security is not (just) virtualization security,” in *Proceedings of the ACM Cloud Computing Security Workshop*, November 2010. 3.3.3
- [37] CLARK, D. D. and WILSON, D. R., “A comparison of commercial and military computer security policies,” in *Proceedings of the 1987 IEEE Symposium on Computer Security and Privacy*, p. 184, 1987. 2.1.1
- [38] CODEWEAVERS, “WineHQ: Run Windows applications on Linux, BSD and Mac OS X.” <http://www.winehq.org/>. 5.5.2
- [39] COLP, P., “VM Snapshots for Xen.” <http://cs.ubc.ca/~pjcolp/>, April 2010. 6.4
- [40] COMMISSION OF THE EUROPEAN COMMUNITIES, *Information Technology Security Evaluation Criteria (ITSEC)*. Commission of the European Communities, com(90) 314 ed., 1991. 2.1.1
- [41] COMMITTEE ON NATIONAL SECURITY SYSTEMS, *National Information Assurance (IA) Glossary*, CNSS Instruction No. 4009 ed., June 2006. 2.1.1
- [42] COMMON CRITERIA RECOGNITION AGREEMENT, “The common criteria portal.” <http://www.commoncriteriaportal.org>. 2.1.1
- [43] CORBATO, F. J. and VYSSOTSKY, V. A., “Introduction and overview of the multics system,” in *Proceedings of the Fall Joint Computer Conference (FJCC)*, 1965. 2.1.1

- [44] COZZIE, A., STRATTON, F., XUE, H., and KING, S. T., “Digging for data structures,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2008. 2.4, 4.1, 4.2.4
- [45] CUI, W., KATZ, R. H., and TIAN TAN, W., “Design and Implementation of an Extrusion-based Break-In Detector for Personal Computers,” in *Proc. of the Annual Computer Security Applications Conference*, 2005. 5.1, 5.2.1
- [46] DAGON, D., ANTONAKAKIS, M., DAY, K., LUO, X., LEE, C. P., and LEE, W., “Recursive DNS Architectures and Vulnerability Implications,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009. 1.1
- [47] DALEY, R. C. and NEUMANN, P. G., “A general purpose file system for secondary storage,” in *Proceedings of the Fall Joint Computer Conference (FJCC)*, 1965. 2.1.1
- [48] DAVID, E. E. and FANO, R. M., “Some thoughts about the social implications of accessible computing,” in *Proceedings of the Fall Joint Computer Conference (FJCC)*, 1965. 2.1.1
- [49] DESCLAUX, F. and KORTCHINSKY, K., “Vanilla Skype,” in *Recon*, 2006. 6.2.3
- [50] DINABURG, A., ROYAL, P., SHARIF, M., and LEE, W., “Ether: Malware analysis via hardware virtualization extensions,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008. 3.4.4.3
- [51] DoD, “Trusted computer system evaluation criteria,” Tech. Rep. DoD 5200.28-STD, Department of Defense, 1985. 2.1.1, 2.1.1
- [52] DOLAN-GAVITT, B., “The VAD tree: A process-eye view of physical memory,” in *Proceedings of the Digital Forensic Research Workshop (DFRWS)*, August 2007. 2.4
- [53] DOLAN-GAVITT, B., “Forensic analysis of the windows registry in memory,” in *Submitted to DFRWS*, 2008. 2.4
- [54] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., and GIFFIN, J., “Robust signatures for kernel data structures,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009. 2.4
- [55] DREGER, H., FELDMANN, A., MAI, M., PAXSON, V., and SOMMER, R., “Dynamic application-layer protocol analysis for network intrusion detection,” in *Proceedings of the USENIX Security Symposium*, 2006. 5.5.3
- [56] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M., and CHEN, P. M., “Revirt: Enabling intrusion analysis through virtual-machine logging and replay,” in *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2002. 2.2.2, 3.3.3
- [57] EBAY, INC., “Presentation on Q1 2009 earning report of Ebay Inc.” <http://www.slideshare.net/earningreport/presentation-on-q1-2009-earning-report-of-ebay-inc>. 6.2.3

- [58] ELSON, J. and CERPA, A., “RFC 3507 - Internet Content Adaptation Protocol (ICAP).” <http://www.ietf.org/rfc/rfc3507.txt>. 6.2.2.3
- [59] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., and XIAO, C., “The daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, vol. 69, pp. 35–45, December 2007. 2.1.1
- [60] FAIGIN, D. P., LINDELL, A. M., PORRAS, P. A., TAPPER, D. R., and MILLEN, J. K., “Final evaluation report: Gemini trusted network processor,” Tech. Rep. S-241,887/34-94, National Computer Security Center, 1995. 2.1.1
- [61] FEIERTAG, R. J. and NEUMANN, P., “The foundations of a provably secure operating system (PSOS),” in *Proceedings of the National Computer Conference*, pp. 329 – 334, AFIPS Press, 1979. 2.1.1
- [62] FORD AEROSPACE, “Secure Minicomputer Operating System (KSOS), Phase 1: Design of the Department of Defense Kernelized Secure Operating System,” tech. rep., Western Development Laboratories Division, 1978. 2.1.1
- [63] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., and LONGSTAFF, T. A., “A sense of self for unix processes,” in *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, May 1996. 5.2.2
- [64] FRAIM, L. J., “Scomp: A solution to the multilevel security problem,” *IEEE Computer*, vol. 16, pp. 26–34, July 1983. 2.1.1
- [65] FRASER, T., EVENSON, M. R., and ARBAUGH, W. A., “VICI – Virtual Machine Introspection for Cognitive Immunity,” in *Proceedings of the Annual Computer Security Applications Conference*, 2008. 2.2.2, 7.1
- [66] FREED, N. and BORENSTEIN, N., “RFC 2045 - Multipurpose Internet Mail Extensions (MIME) Part One.” <http://www.ietf.org/rfc/rfc2045.txt>. 6.2.1.3
- [67] GAO, D., REITER, M., and SONG, D., “Behavioral distance for intrusion detection,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005. 2.1.1
- [68] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., and BONEH, D., “Terra: A virtual machine-based platform for trusted computing,” in *Proceedings of ACM Symposium on Operating Systems Principles*, 2003. 2.1.1, 3.3.3
- [69] GARFINKEL, T. and ROSENBLUM, M., “A virtual machine introspection based architecture for intrusion detection,” in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003. 2.2.2, 2.2.2, 3.1, 3.2.1, 3.3.3, 5.3.2, 5.4
- [70] GARFINKEL, T. and ROSENBLUM, M., “When virtual is harder than real: Security challenges in virtual machine based computing environments,” in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HOTOS-X)*, May 2005. 2.3

- [71] GASSER, M., *Building A Secure Computer System*. Van Nostrand Reinhold, 1988. 2.1.1
- [72] GLASER, E. L., COULEUR, J. F., and OLIVER, G. A., “System design of a computer for time-sharing applications,” in *Proceedings of the Fall Joint Computer Conference (FJCC)*, 1965. 2.1.1
- [73] GOLDBERG, R., “Survey of virtual machine research,” *IEEE Computer Magazine*, vol. 7, pp. 34 – 45, June 1974. 2.3
- [74] GOLDBERG, R. P., *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973. 2.3
- [75] GRAWROCK, D., *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009. 5.2.1
- [76] GU, G., PERDISCI, R., ZHANG, J., and LEE, W., “BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection,” in *Proceedings of the USENIX Security Symposium*, 2008. 5.1
- [77] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., and LEE, W., “BotHunter: Detecting malware infection through ids-driven dialog correlation,” in *Proceedings of the USENIX Security Symposium*, 2007. 5.1
- [78] GUMMADI, R., BALAKRISHNAN, H., MANIATIS, P., and RATNASAMY, S., “Not-a-Bot (NAB): Improving Service Availability in the Face of Botnet Attacks,” in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. 5.1, 5.2.1, 5.3.1
- [79] HAND, S., WARFIELD, A., and FRASER, K., “Are virtual machine monitors microkernels done right?,” in *Proceedings of the 2005 Workshop on Hot Topics in Operating Systems*, 2005. 2.3
- [80] HAY, B. and NANCE, K., “Forensics examination of volatile system data using virtual introspection,” *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 74 – 82, April 2008. 2.2.2
- [81] HEDLEY, J., “jsoup: Java html parser.” <http://jsoup.org/>, April 2010. 6.2.2.2
- [82] HEISER, G., UHLIG, V., and LEVASSEUR, J., “Are virtual-machine monitors microkernels done right?,” *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 95 – 99, January 2006. 2.3
- [83] HOGLUND, G., *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2005. 3.6.1.1
- [84] HOHMUTH, M., PETER, M., HARTIG, H., and SHAPIRO, J. S., “Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors,” in *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004. 2.3

- [85] HOLLINGWORTH, D. and REDMOND, T., “Enhancing operating system resistance to information warfare,” in *Proceedings of the 21st Century Military Communications Conference (MILCOMM 2000)*, vol. 2, pp. 1037 – 1041, 2000. 2.2.1
- [86] JAEGER, T., *Operating System Security*. Morgan & Claypool, 2008. 2.1.1
- [87] JIANG, X. and WANG, X., ““Out-of-the-box” Monitoring of VM-based High-Interaction Honeypots,” in *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007. 2.2.2, 3.2.1
- [88] JIANG, X., XU, D., and WANG, X., “Stealthy Malware Detection Through VMM-Based “Out-of-the-Box” Semantic View Reconstruction,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, October 2007. 2.2.2, 3.1
- [89] JONES, S. T., ARPACI-DUSSEAU, A. C., and ARPACI-DUSSEAU, R. H., “Antfarm: Tracking processes in a virtual machine environment,” in *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ’06)*, June 2006. 2.2.2, 3.1
- [90] JONES, S. T., ARPACI-DUSSEAU, A. C., and ARPACI-DUSSEAU, R. H., “VMM-based Hidden Process Detection and Identification using Lycosid,” in *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2008. 2.2.2
- [91] JOSHI, A., KING, S. T., DUNLAP, G. W., and CHEN, P. M., “Detecting past and present intrusions through vulnerability-specific predicates,” in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pp. 1–15, October 2005. 2.2.2, 3.3.3, 5.3.2, 5.4
- [92] JUNG, C. and CLARK, N., “DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage,” in *Proceedings of the International Symposium on Microarchitecture*, 2009. 6.5
- [93] KARGER, P. A., “Multi-level security requirements for hypervisors,” in *Annual Computer Security Applications Conference (ACSAC)*, pp. 267 – 275, December 2005. 2.3
- [94] KARGER, P. A. and SCHELL, R. R., “Multics security evaluation: Vulnerability analysis,” Tech. Rep. ESD-TR-74-193, ESD/AFSC, 1974. 2.1.1
- [95] KARGER, P. A. and SCHELL, R. R., “Thirty years later: Lessons from the multics security evaluation,” in *Proceedings of the Annual Computer Security Applications Conference*, 2002. 2.1.1
- [96] KARGER, P. A., ZURKO, M. E., BONIN, D. W., MASON, A. H., and KAHN, C. E., “A Retrospective on the VAX VMM Security Kernel,” *IEEE Transactions on Software Engineering*, vol. 17, pp. 1147 – 1165, November 1991. 2.1.1

- [97] KELEM, N. L. and FEIERTAG, R. J., “A separation model for virtual machine monitors,” in *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pp. 78 – 86, 1991. 2.3
- [98] KING, S. T. and CHEN, P. M., “Backtracking intrusions,” in *Proceedings of the Symposium on Operating System Principles*, October 2003. 2.2.2
- [99] KING, S. T. and CHEN, P. M., “Backtracking intrusions,” *ACM Transactions on Computer Systems*, vol. 23, pp. 51 – 76, 2005. 2.2.2
- [100] KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., and LORCH, J. R., “Subvirt: Implementing malware with virtual machines,” in *IEEE Symposium on Security and Privacy*, 2006. 3.3.3
- [101] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., and WANG, X., “Effective and efficient malware detection at the end host,” in *Proceedings of the USENIX Security Symposium*, 2009. 1.1
- [102] KORNBLUM, J. D., “Using every part of the buffalo in windows memory analysis,” *Digital Investigation*, vol. 4, pp. 24 – 29, March 2007. 2.4
- [103] KOURAI, K. and CHIBA, S., “Hyperspector: Virtual distributed monitoring environments for secure intrusion detection,” in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005. 2.2.2, 3.1
- [104] LAMPSON, B. W., “A note on the confinement problem,” *Communications of the ACM*, vol. 16, pp. 613 – 615, October 1973. 2.3
- [105] LASKO, T. A. and HAUSER, S. E., “Approximate string matching algorithms for limited-vocabulary OCR output correction,” in *Proceedings of SPIE*, vol. 4307, pp. 232 – 240, 2001. 6.2.1.2
- [106] LEE, W. and STOLFO, S. J., “A framework for constructing features and models for intrusion detection systems,” *ACM Transactions on Information and System Security*, vol. 3, November 2000. 5.2.2
- [107] LEVENSHTAIN, V., “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” *Soviet Physics Doklady*, vol. 10, p. 707, 1966. 6.2.1.2
- [108] LITTY, L., “Hypervisor-based intrusion detection,” Master’s thesis, University of Toronto, 2005. 2.2.2
- [109] LOVE, R., *Linux Kernel Development*. Novell Press, 2nd ed., 2005. 3.5.1.3
- [110] MADNICK, S. E. and DONOVAN, J. J., “Application and analysis of the virtual machine approach to information system security and isolation,” in *Proceedings of the Workshop on Virtual Computer Systems*, pp. 210 – 224, March 1973. 2.1.1, 2.3

- [111] MARTIN, W., WHITE, P., TAYLOR, F. S., and GOLDBERG, A., “Formal construction of the mathematically analyzed separation kernel,” in *Proceedings of the IEEE International Conference on Automated Software Engineering*, 2000. 2.1.1
- [112] McCUNE, J. M., BERGER, S., CACERES, R., JAEGER, T., and SAILER, R., “Shamon – a system for distributed mandatory access control,” in *22nd Annual Computer Security Applications Conference (ACSAC)*, December 2006. 2.1.1
- [113] MEUSHAW, R. and SIMARD, D., “Nettop: A network on your desktop,” *Tech Trend Notes (National Security Agency)*, vol. 9, pp. 3 – 11, Fall 2000. 2.1.1
- [114] MICROSOFT, “Debugging tools for windows.” <http://www.microsoft.com/whdc/devtools/debugging/default.msp>. 4.2.3
- [115] MICROSOFT, “Kernel memory space analyzer.” <http://www.microsoft.com>. 4.2.3
- [116] MITTIG, K., “Greasyspoon: Scripting factory for core network services.” <http://greasyspoon.sourceforge.net/>, April 2010. 6.2.2.3
- [117] MOSER, A., KRUEGEL, C., and KIRDA, E., “Exploring multiple execution paths for malware analysis,” in *Proceedings of the IEEE Symposium of Security and Privacy*, May 2007. 3.2.1
- [118] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S. D., and LEVY, H. M., “SpyProxy: execution-based detection of malicious web content,” in *Proceedings of the USENIX Security Symposium*, pp. 1–16, 2007. 6.2.2.4
- [119] NATIONAL SECURITY AGENCY, “Defense in Depth: A practical strategy for achieving Information Assurance in today’s highly networked environments.” http://www.nsa.gov/ia/_files/support/defenseindepth.pdf. 5.2.2
- [120] NCSC, “Trusted network interpretation of the trusted computer system evaluation criteria,” Tech. Rep. NCSC-TG-005, Department of Defense, 1987. 2.1.1
- [121] NCSC, “Trusted database mangement system interpretation of the trusted computer system evaluation criteria,” Tech. Rep. NCSC-TG-021, Department of Defense, 1991. 2.1.1
- [122] OBERHEIDE, J., COOKE, E., and JAHANIAN, F., “CloudAV: N-Version Antivirus in the Network Cloud,” in *Proceedings of the USENIX Security Symposium*, 2008. 1.1, 5.2.2
- [123] OSSANNA, J. F., MIKUS, L., and DUNTEN, S. D., “Communications and input/output switching in a multiplex computing system,” in *Proceedings of the Fall Joint Computer Conference (FJCC)*, 1965. 2.1.1
- [124] PAXSON, V., “Bro: A System for Detecting Network Intruders in Real-Time,” in *Proceedings of the USENIX Security Symposium*, 1998. 5.2.2

- [125] PAYNE, B. D., CARBONE, M., and LEE, W., “Secure and flexible monitoring of virtual machines,” in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pp. 385 – 397, December 2007. 3.1, 3.2.1, 4.2.1, 7.1
- [126] PAYNE, B. D., CARBONE, M., SHARIF, M., and LEE, W., “Lares: An architecture for secure active monitoring using virtualization,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008. 3.3.3, 5.3.2, 5.4, 7.1
- [127] PAYNE, B. D., SAILER, R., CACERES, R., PEREZ, R., and LEE, W., “A layered approach to simplified access control in virtualized systems,” *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 12 – 19, July 2007. 2.1.1, 5.5
- [128] PERDISCI, R., DAGON, D., LEE, W., FOGLA, P., and SHARIF, M., “Misleading worm signature generators using deliberate noise injection,” in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006. 1.1
- [129] PETRONI, JR., N. L., FRASER, T., MOLINA, J., and ARBAUGH, W. A., “Copilot - a coprocessor-based kernel runtime integrity monitor,” in *Proceedings of the 13th USENIX Security Symposium*, August 2004. 2.2.1
- [130] PETRONI, JR., N. L., FRASER, T., WALTERS, A., and ARBAUGH, W. A., “An architecture for specification-based detection of semantic integrity violations in kernel dynamic data,” in *Proceedings of the USENIX Security Symposium*, 2006. 2.1.1, 3.6.2.3
- [131] PETRONI, JR., N. L. and HICKS, M., “Automated detection of persistent kernel control-flow attacks,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, October 2007. 1.1, 2.1.1, 3.6.2.3, 4.2.4
- [132] PETRONI, JR., N. L., WALTERS, A., FRASER, T., and ARBAUGH, W. A., “Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory,” *Digital Investigation*, vol. 3, pp. 197 – 210, December 2006. 2.4, 4.2.4
- [133] PEW INTERNET, “Daily Internet Activities, 2000–2009.” <http://www.pewinternet.org/Trend-Data/Daily-Internet-Activities-20002009.aspx>. 6.1
- [134] QUMRANET, INC., “KVM: Kernel-based Virtualization Driver,” 2006. 2.3
- [135] RABINER, L. and JUANG, B.-H., *Fundamentals of Speech Recognition*. Prentice Hall, 1993. 4.3.2
- [136] RESIG, J., “Envjs.” <http://env-js.appspot.com/>, April 2010. 6.2.2.2, 6.2.2.4
- [137] RESNICK, P., “RFC 2822 - Internet Message Format.” <http://www.ietf.org/rfc/rfc2822.txt>. 6.2.1.3
- [138] ROBIN, J. S. and IRVINE, C. E., “Analysis of the Intel pentium’s ability to support a secure virtual machine monitor,” in *Proceedings of the 9th USENIX Security Symposium*, August 2000. 2.3, 3.2.2.1

- [139] ROSENBLUM, N. E., COOKSEY, G., and MILLER, B. P., “Virtual machine-provided context sensitive page mappings,” in *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2008. 2.1.1
- [140] RUEDA, S., SREENIVASAN, Y., and JAEGER, T., “Flexible security configuration for virtual machines,” in *Proceedings of the 2nd ACM workshop on Computer security architectures*, 2008. 2.3
- [141] RUSHBY, J. M., “Proof of separability: A verification technique for a class of security kernels,” *Lecture Notes in Computer Science*, vol. 137, pp. 352 – 357, April 1982. 2.1.1, 2.3
- [142] RUSHBY, J., “Design and verification of secure systems,” *ACM Operating Systems Review*, vol. 15, no. 5, pp. 12–21, 1981. 2.1.1
- [143] RUSSINOVICH, M. E. and SOLOMON, D. A., *Microsoft Windows Internals*. Microsoft Press, 4th ed., 2004. 3.5.1.3, 3.8.1
- [144] RUTKOWSKA, J., “Rootkit hunting vs. compromise detection,” in *Proceedings of Black Hat Federal*, 2006. 3.6.1.1
- [145] RUTKOWSKA, J., “Subverting vista kernel for fun and profit,” in *Proceedings of Black Hat USA*, 2006. 3.3.3, 6.5
- [146] SAILER, R., JAEGER, T., VALDEZ, E., CACERES, R., PEREZ, R., BERGER, S., GRIFFIN, J., and VAN DOORN, L., “Building a MAC-based security architecture for the xen opensource hypervisor,” in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, December 2005. 2.3, 3.2.2.3
- [147] SAILER, R., ZHANG, X., JAEGER, T., and VAN DOORN, L., “Design and Implementation of a TCG-based Integrity Measurement Architecture,” in *Usenix Security Symposium*, 2004. 2.1.1
- [148] SALTZER, J. H. and SCHROEDER, M. D., “The protection of information in computer systems,” *Communications of the ACM*, vol. 17, July 1974. 2.3, 3.3.1
- [149] SAYDJARI, O. S., BECKMAN, J. M., and LEAMAN, J. R., “LOCK trek: Navigating uncharted space,” in *Proceedings of the Symposium on Security and Privacy*, 1989. 2.1.1
- [150] SCHILLER, W. L., “The design and specification of a security kernel for the PDP-11/45,” Tech. Rep. MTR-2934, Mitre, 1975. 2.1.1
- [151] SCHUSTER, A., “Pool allocations as an information source in windows memory forensics,” in *International Conference on IT-Incident Management & IT-Forensics*, October 2006. 2.4
- [152] SCHUSTER, A., “Searching for processes and threads in microsoft windows memory dumps,” in *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS)*, August 2006. 2.4, 4.1, 4.2.2, 4.3.5.1

- [153] SEKAR, R., “On Preventing Intrusions by Process Behavior Monitoring,” in *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999. 5.2.2
- [154] SEKAR, R. and UPPULURI, P., “Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications,” in *Proceedings of the USENIX Windows NT Symposium*, 1999. 5.2.2
- [155] SESHADRI, A., LUK, M., QU, N., and PERRIG, A., “SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes,” in *Proceedings of the ACM Symposium on Operating System Principles*, 2007. 2.1.1, 3.1, 3.3.2, 3.6.2.3
- [156] SHARIF, M., LANZI, A., GIFFIN, J., and LEE, W., “Automatic reverse engineering of malware emulators,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009. 6.5
- [157] SHNEIDERMAN, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, third ed., 1998. 6.4
- [158] SHOCKLEY, W. R. and SCHELL, R. R., “TCB Subsets for Incremental Evaluation,” in *Proceedings of the 3rd Aerospace Computer Security Conference*, pp. 131 – 139, December 1987. 2.3
- [159] SLEATOR, D. D. and TARJAN, R. E., “Self-Adjusting Binary Search Trees,” *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985. 6.2.1.2
- [160] SMITH, R., “An Overview of the Tesseract OCR Engine,” in *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*, pp. 629–633, 2007. 6.2.1.2
- [161] SMITH, S. W., *Trusted Computing Platforms: Design and Applications*. Springer, 2005. 3.3.3, 5.3.2
- [162] SRIVASTAVA, A. and GIFFIN, J., “Tamper-resistant, application-aware blocking of malicious network connections,” in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008. 5.2.2
- [163] STEINHAUS, H., “Sur la division des corp materiels en parties,” *Bulletin L'Academie Polonaise des Science*, vol. C1 III, no. IV, pp. 801 – 804, 1956. 4.3.2
- [164] SZOR, P., “Memory Scanning Under Windows NT,” in *Proceedings of the 9th International Virus Bulletin Conference*, October 1999. 2.4
- [165] SZOR, P., *The Art of Computer Virus Research and Defense*. Symantec Press, 2005. 3.6.1.1, 5.2.2
- [166] THE MOZILLA PROJECT, “Rhino: Javascript for java.” <http://www.mozilla.org/rhino/>. 6.2.2.4
- [167] THE MOZILLA PROJECT, “Spidermonkey (javascript-c) engine.” <http://www.mozilla.org/js/spidermonkey/>. 6.2.2.4

- [168] TREND MICRO, “Worm/agobot.hd, also known as w/32 agobot.f.” <http://www.trendmicro.com/vinfo/virusencyclo/>. 3.6.1.1
- [169] VARIAN, M., “VM and the VM Community: Past, Present, and Future.” <http://www.princeton.edu/melinda/25paper.pdf>, 1997. 2.3, 2.3
- [170] VITERBI, A., “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Transactions on Information Theory*, vol. 13, pp. 260 – 269, April 1967. 4.3.3
- [171] VLECK, T. V., “Multics website.” <http://www.multicians.org>. 2.1.1
- [172] VMWARE, INC., “Vmware milestones.” <http://www.vmware.com/company/mediaresource/milestones.html>. 2.3
- [173] VMWARE, INC., “VMWare VMsafe security technology.” <http://www.vmware.com/technology/security/vmsafe.html>. 3.4.4.3, 7.1, 7.2
- [174] VON AHN, L., BLUM, M., HOPPER, N., and LANGFORD, J., “CAPTCHA: Using Hard AI Problems for Security,” in *Advances in Cryptology — EUROCRYPT*, 2003. 5.2.1
- [175] VYSSOTSKY, V. A., CORBATO, F. J., and GRAHAM, R. M., “Structure of the multics supervisor,” in *Proceedings of the Fall Joint Computer Conference (FJCC)*, 1965. 2.1.1
- [176] WALTER, S., “ProxSMTP: An SMTP Filter.” <http://memberwebs.com/stef/software/proxsmtp/>. 6.2.1.3
- [177] WALTERS, A., “The volatility framework: Volatile memory artifact extraction utility framework.” <https://www.volatilesystems.com/default/volatility/>. 2.4, 4.2.1, 4.3.5.1
- [178] WALTERS, A., “FATKit: Detecting malicious library injection and upping the “anti,”” July 2006. 4.2.4
- [179] WANG, Y.-M., BECK, D., VO, B., ROUSSEV, R., and VERBOWSKI, C., “Detecting stealth software with strider ghostbuster,” in *International Conference on Dependable Systems and Networks (DSN’05)*, pp. 368 – 377, 2005. 2.2.2
- [180] WEI, J., PAYNE, B. D., GIFFIN, J., and PU, C., “Soft-timer driven transient kernel control flow attacks and defense,” in *Proceedings of the Annual Computer Security Applications Conference*, December 2008. 7.1
- [181] WESSELS, D., NORDSTRÖM, H., ROUSSKOV, A., CHADD, A., COLLINS, R., SERASSIO, G., WILTON, S., and FRANCESCO, C., “Squid: Optimising web delivery.” <http://www.squid-cache.org/>. 5.5.3, 6.2.2.3
- [182] WHITAKER, A., COX, R. S., SHAW, M., and GRIBBLE, S. D., “Constructing services with interposable virtual hardware,” March 2004. 2.3

- [183] XU, M., JIANG, X., SANDHU, R., and ZHANG, X., “Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection,” in *Proceedings of the Symposium on Access control Models and Technologies (SACMAT)*, June 2007. 2.2.2, 3.6.2.3
- [184] YANG, J. and SHIN, K., “Using hypervisor to provide application data secrecy on a per-page basis,” in *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, March 2008. 2.1.1
- [185] YIN, H., LIANG, Z., and SONG, D., “HookFinder: Identifying and understanding malware hooking behaviors,” in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2008. 3.6.1.1
- [186] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., and KIRDA, E., “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the ACM Conference of Computer and Communication Security*, 2007. 3.2.1
- [187] ZANDY, V. and RIDGE, D., “First-class C Contexts in Cinquecento,” tech. rep., IDA Center for Computing Sciences, 2008. 7.2
- [188] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., and SAILER, R., “Secure coprocessor-based intrusion detection,” in *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002. 2.2.1