ROBUST AND SECURE MONITORING AND ATTRIBUTION OF MALICIOUS BEHAVIORS

A Thesis
Presented to
The Academic Faculty

by

Abhinav Srivastava

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

Georgia Institute of Technology
August 2011

ROBUST AND SECURE MONITORING AND ATTRIBUTION OF MALICIOUS BEHAVIORS

Approved by:

Professor Jonathon Giffin, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Wenke Lee
School of Computer Science
Georgia Institute of Technology

Professor Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Patrick Traynor
School of Computer Science
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Date Approved: 13 June 2011

To my elder brother, Anurag

for encouraging me to pursue this journey, and

to my wife, Neha

for helping me successfully complete it.

ACKNOWLEDGEMENTS

No document of this size can be prepared without the support of numerous people. While it is difficult to mention everyone, I would like to take this opportunity to thank some individuals who really made this work possible.

I express my gratitude to my advisor Jon Giffin for guiding my dissertation research. His constant guidance, encouragement, and vision made this journey possible. The tireless meetings and brain-storming sessions with him taught me deep concepts of computer security. He explained me the importance of conducting systematic research, writing scientific documents, and presenting my research in conferences and other public venues. He also facilitated me see the difference between software development and research. Jon helped me grow as a scientist. I wish I could ever repay him the debt I owe.

I am thankful to my dissertation committee members Mustaque Ahamad, Dough Blough, Wenke Lee, and Patrick Traynor for reading the thesis and providing insightful comments. Their valuable suggestions have improved the quality of this document.

Professors and teachers teach you academics, but what truly make a graduate school memorable is its students. During my stay in Georgia Tech, I collaborated with Brendan Dolan-Gavit, Ikpeme Erete, Daisuke Mashima, Andrea Lanzi, and Kapil Singh. I also interacted with Manos Antonakakis, Martim Carbone, Long Lu, Roberto Perdisci, Monirul Sharif, and Junjie Zhang. I thank all of them for collaborating with me, providing early feedback on my research, proof-reading my research papers, and making my stay memorable and worthwhile. I am also thankful to GTISC, Mary Claire, and Alfreda Barrow for providing all the help and facilities needed to carry out my research.

I would like to give a big and special thanks to my parents Umesh and Kiran, who gave everything to their children. I am here because of their unconditional love, hard work, and prayers. I am also deeply indebted to my elder brother Anurag, who always encouraged me to strive for the best and kept faith in me during my tough times. My younger brother Anuj always reminded me to do well to set better examples for him as my elder brother did for me. I am also thankful to my second beautiful and loving family, Arvind and Neena for believing in me and allowing me to marry their wonderful daughter. I also take this opportunity to express my gratitude to Shivani, Ankita, Ankit, Ketan, and Aarushi for being a part of my family and bringing joy to my life.

Last but not least, I owe a big thanks to my wife Neha, who was right by my side during all my ups and downs. She was considerate when I worked late or on weekends due to deadlines. It is hard to imagine completing this journey without her. Her constant love and support is the only thing that kept me going, and I look forward to spend the rest of my life with her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Worldwide computer systems continue to execute malicious software that degrades the systems' performance and consumes network capacity by generating high volumes of unwanted traffic. Network-based detectors can effectively identify machines participating in the ongoing attacks by monitoring the traffic to and from the systems. But, network detection alone is not enough; it does not improve the operation of the Internet or the health of other machines connected to the network. We must identify malicious code running on infected systems, participating in global attack networks.

This dissertation describes a robust and secure approach that identifies malware present on infected systems based on its undesirable use of network. Our approach, using virtualization, attributes malicious traffic to host-level processes responsible for the traffic. The attribution identifies on-host processes, but malware instances often exhibit parasitic behaviors to subvert the execution of benign processes.

We then augment the attribution software with a host-level monitor that detects parasitic behaviors occurring at the user- and kernel-level. User-level parasitic attack detection happens via the system-call interface because it is a non-bypassable interface for user-level processes. Due to the unavailability of one such interface inside the kernel for drivers, we create a new driver monitoring interface inside the kernel to detect parasitic attacks occurring through this interface.

Our attribution software relies on a guest kernel's data to identify on-host processes. To allow secure attribution, we prevent illegal modifications of critical kernel data from kernel-level malware. Together, our contributions produce a unified research outcome – an improved malicious code identification system for user- and kernel-level malware.

CHAPTER I

INTRODUCTION

The cybercrime industry is growing at a fast pace. For example, Sophos received 60,000 new malware samples every day in the first half of 2010 [137]. This number is 50% higher than the previous year's number gathered around the same time [137]. Malicious software instances, such as worms, viruses, bots, spyware, and adware, continue to attack critical infrastructures, degrade systems' performance delivered to their legitimate users, and consume network capacity by generating high volumes of unwanted traffic. Along with activities such as spam generation, denial-of-service attacks, and malware propagation, malware authors also launch targeted espionage against countries and organizations [79, 171].

Given this high growth in malware, security software attempts to detect malicious programs in a timely manner so that their malicious behaviors can be contained and infected machines can be remediated. However, security tools continue to fail to detect malicious code present on infected systems. Another report shows that popular anti-virus (AV) tools can only detect 18.9% of new malware attacks [27]. This leaves a wide gap between attacks encountered and attacks detected.

The visible effects of current attacks against software regularly manifest first as suspicious network traffic. This is due to the monetary gains involved in controlling large networks for botnet, spam, and denial of service attacks [144]. For example, the conficker worm infected millions of machines in less than 24 hours to create a botnet [149]. To protect critical infrastructure, network administrators usually deploy intrusion detection systems and firewalls in networks. Network-based intrusion detection systems have a global view of a network as they can see incoming and

outgoing traffic for all hosts within the network. This network-centric design makes them attractive solutions to detect co-ordinated attacks such as a botnet in which multiple infected machines participate against the protected networks and Internet [56, 57, 111]. Moreover, these security tools are tamper-resistant as they work on networks and are completely isolated from infected host systems.

Network-level security solutions can only pinpoint individual infected machines i.e. location of the malicious code. They fail to identify the actual malicious code present on infected systems responsible for suspicious traffic. This is due to the fact that host-level malicious code execution information is not visible on networks. Identifying malicious code is useful in many scenarios such as malware detection, malware analysis, and remediation of infected systems. On-host malicious code corresponding to malware instances that send or receive malicious traffic can only be identified if we attribute observed network behaviors to the responsible malicious code installed on an infected system. The attribution that identifies the malicious code is *fine-grained* as compared to coarse-grained attribution that identifies only infected machines. To provide such fine-grained attribution of malicious network traffic to the actual on-host malicious code, we need to find out what is happening inside an infected system.

The information not visible to network-level security solutions can be extracted using host-level security utilities. Host-level security tools have a local, but complete view of a single system. They can detect malicious code either by employing signatures of known malware or by monitoring the runtime execution of processes. For example, anti-virus software running on end users' systems identifies malicious code by matching signatures [93, 109], while host-based system-call monitors detect attacks by observing anomalies in the execution of benign software under the attack [49–51, 61, 132].

As attackers and defenders are in an arms race, malicious programs exhibit complex behaviors on end users' systems to avoid detection from security software. To

keep their presence obscure, malware instances often subvert the normal execution of benign processes by modifying their in-memory code image. We term this kind of behaviors as *parasitic behaviors*. By using parasitic activities, malware forces legitimate programs to carry out illegitimate activities without raising any suspicion. To make the matter worse, any heuristic-based malware detection system that monitors system calls or detects code injection attacks may produce false positives. For example, DLL injection is used by the Microsoft Visual Studio debugger to monitor processes under development. Likewise, the Google toolbar injects code into explorer.exe (the Windows graphical file browser) to provide Internet search from the desktop. Hence, just by observing the presence of parasitic activities, it is hard to classify them as attacks.

Further attempts to evade existing security utilities include attainment of the highest privilege layer in a system. Traditional malware used to attack user-level applications. This is not true anymore. Today, we are facing the emerging threat of kernel-level malware [135]. Malicious programs, often called rootkits, install themselves in the kernel in the form of drivers and modify the kernel's code and data. Since rootkits run as the most privileged code in the system, traditional host-based security software fails to detect these attacks. Moreover, rootkits often disable current host-based security software executing on infected systems.

This dissertation tackles the malicious code identification problem by providing the best of both network and host security solutions. The key idea is to correlate network-level packet information with host-level software execution knowledge so that we can attribute malicious network traffic to host-level malicious code that is responsible for the traffic. Taken alone, either approach will have diminished utility in the presence of typical attacks or normal workloads. Network-based detection can identify an infected system but cannot provide fine-grained process-specific information.

Host-based detection can identify occurrences of parasitism, but it cannot differentiate malicious parasites from benign symbiotes. For example, debugging software and other benign software, such as the Google toolbar, use DLL injection for legitimate purposes. These observations are critical: A process sending or receiving malicious network traffic may not itself be malware, and a process injecting code into another process may not be malicious. Only by linking injection with subsequent malicious activity observed at the network (or other) layer can we correctly judge the activity at the host.

Our approach correlates network-level events with host-level activities, so it applies exclusively to attacks that send or receive detectably suspicious traffic. Hence, we assume that the presence of malware can be detected by its network behaviors. Our approach has several benefits over previous approaches: it does not require the use of signature matching on a host. Though signature matching schemes can be used on the network side to detect hidden attacks, nowadays malicious software connects to malicious servers, sends spams in bulk, and initiates DDoS attacks, which is difficult to hide. Further, fine-grained malicious code identification creates the foundation for surgical remediation. The coarse-grained information provided by the network-level software permits only coarse-grained responses: an administrator could excise an infected system from the network, possibly for re-imaging. Unfortunately, in many common use scenarios, complete disk sanitization results in intolerable losses of critical data not stored elsewhere, even though that data may not have been affected by the infection. The fine-grained attribution information changes these brutal remediation techniques by providing a means to appropriately attribute malicious behavior to malicious software. By gaining a better understanding of a malware infection on a system, we can offer opportunities for surgical response.

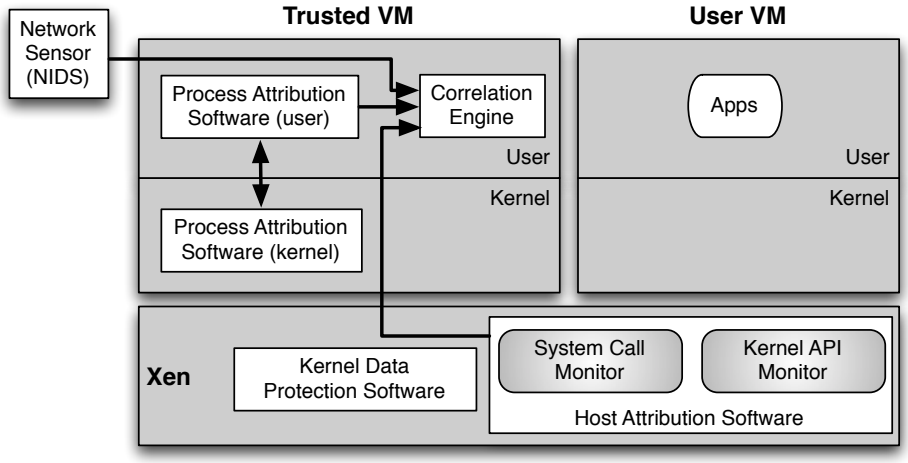


Figure 1: Complete system architecture for identifying malicious code present on infected systems.

1.1 Challenges and Contributions

This thesis presents a complete system that identifies malicious code present on infected machines. Figure 1 shows the complete architecture of our system. We leverage virtualization technologies to realize our architecture. This design choice is due to the fact that virtualization allows security software to be run in isolation from malicious software and provides capabilities to monitor infected machines. Further, virtualization features are also well supported by the recent commodity hardware. We deploy our software components inside the trusted virtual machines and hypervisor, and run infected systems inside untrusted user virtual machines. As shown in Figure 1, we also use one aggregator called the *correlation engine* that gathers information from other software components to identify all the malicious code present inside infected virtual machines. Our architecture includes a network intrusion detection system (NIDS) that runs on the perimeter of the network and detects malicious traffic; any off-the-shelf NIDS can be used for this purpose. After detecting the malicious traffic, identifying on-host malicious code requires solutions to multiple challenging and open problems. We present the most important high-level challenges and our contributions

in solving them below.

1.1.1 Process Attribution

To determine the end-point of a malicious connection, we must be able to identify a host-level process that is bound to the connection. This connection-related information is stored inside the user VM's kernel data, however we cannot install any software inside the user VM to extract this information as attackers can easily subvert our security software. We need to extract the information stored in the kernel data in a secure way without deploying any tool inside the compromised VM.

Contributions: We design and develop a process attribution software that correlates network flows with the host-level information to identify processes bound to the flows. As shown in Figure 1, the process attribution software operates from a trusted virtual machine. Though this design protects our process attribution software from attacks running inside the untrusted VM, it introduces a semantic gap [17]. To bridge this gap, we use virtual machine introspection to read the memory state of user VMs.

1.1.2 Process Manipulation

Even if we identify an end-point of a connection by finding out a process that is bound to the connection, it is difficult to know whether the process is a malicious program or a benign program suffering from parasitic attacks. Parasitic attacks may occur from user-level malware instances. To detect these attacks, we model parasitic behaviors in terms of system events.

Contributions: We correctly attribute observed network behaviors to the actual malware responsible for creating those behaviors, even in the presence of parasitic malware that injects code into benign applications. We design and develop a host attribution software that operates from the hypervisor as shown in Figure 1. We first model parasitic behaviors using an automaton to capture runtime execution behaviors. We then intercept system calls invoked by processes from the hypervisor

and match against the automaton to detect user-level parasitic attacks.

1.1.3 Kernel Code Monitoring

Parasitic attacks can also occur from malicious kernel drivers. To be able to detect kernel-level parasitic attacks, we must be able to monitor drivers' execution behaviors. However, there is no hardware enforced monitoring interface inside operating systems that can be used to monitor drivers' execution behaviors.

Contributions: We create a new monitoring interface inside the kernel for drivers to record kernel-level parasitism. Our software builds a new interface by creating distinct virtual memory regions for commodity monolithic kernels and their drivers in the same way that kernels manage distinct memory regions for higher-level application software. We reduce the performance overhead caused due to separate address spaces by efficiently handling control flows spanning the kernel interface barrier via on-demand dynamic binary rewriting and runtime code generation. Figure 1 shows our kernel API monitor inside the host attribution software.

1.1.4 Kernel Data Protection

The attribution of malicious network traffic to host-level processes occurs in a tamper-resistant way, but the attribution software still peeks into the dynamic kernel data of an infected user VM. Malware instances present inside the infected VM may modify the kernel data illegitimately to fool the process attribution software. To be able to correctly perform the correlation, we must protect the kernel data from unauthorized modifications.

Contributions: We design kernel data protection software that is capable of protecting both statically and dynamically allocated kernel data. Our access control software creates protected memory regions within the unified kernel data space. A kernel can then isolate its security-critical data from malicious drivers having low

trust, creating assurance in the critical state. We also show how to optimize kernel memory space layout to reduce the performance overhead due to our protection. Figure 1 shows our data protection software inside the hypervisor.

1.2 Thesis Overview

This thesis presents ***a practical, robust, and secure hypervisor-based architecture that on receiving alerts from network-level security software, identifies the malicious code present on infected systems by attributing the alerts to host-level processes and monitoring their execution behaviors.*** Given that previous research has approached the malicious code identification problem either from the network or host side, we attempt to combine the information present at both places, providing the best of both worlds. To realize this goal, we must adopt a *practical* approach that does not require changes to software and hardware configurations. The proposed system must also be *robust* enough to withstand against both user- and kernel-level malware. Finally, the new system must be able to extract the required information *securely* without directly getting attacked by malware. We meet all the requirements described here. Our architecture is practical as it utilizes hardware virtualization features present in existing commodity systems and does not demand any hardware and software changes. Our system is robust because its protection and monitoring software encompass both user- and kernel-level malware and extract the execution information accurately. Finally, our attribution and monitoring software are secure as they operate from trusted virtual machines and hypervisors.

The rest of the dissertation is organized as follows: we describe background and related works in the area of systems security in Chapter 2. Chapter 3 presents the design, implementation, and evaluation details of the process attribution software that attributes a malicious network connection to a host-level process that is bound

to the connection. Chapter 4 describes our solutions to detect parasitic attacks. We present descriptions of both user- and kernel-level parasitic attacks and describe our defenses and evaluations for user-level parasitic behaviors. Chapter 5 covers details of our defenses for kernel-level parasitic attacks. We present the design and evaluation of a new driver monitoring interface inside the kernel, the detail of our kernel-level monitor, and our approach to overcome the performance challenges imposed by our design. Chapter 6 details our protection of critical kernel-data on which the process attribution software relies on for identifying an end-point of the connection. We present memory access control software that prevents unauthorized drivers from modifying the critical kernel data, describe a memory partitioning approach to reduce the performance overhead due to the access control, and evaluate the effectiveness of our approach. Finally, we conclude with discussions on future research directions in Chapter 7.

CHAPTER II

BACKGROUND AND RELATED WORK

Host-based computer security is a highly active area of research. Previous research has proposed many solutions to improve the security of applications and operating systems. We first present previous research works carried out in the area of host-based intrusion detection in Section 2.1. Then, we describe various monitoring architectures developed over the years and their effectiveness in detecting attacks in Section 2.2. Researchers have also developed approaches in which they correlated information gathered from various sources and utilized it for the attack detection. The detailed description of correlation based approaches is presented in Section 2.3. After focusing on the application-level security, we turn to operating systems' security. In Section 2.4, we capture the evolution of secure operating systems by providing the design of some popular kernels. Finally, we present the background of virtualization and previous works in the area of virtualization-based security in Section 2.5.

2.1 Host-based Intrusion Detection

Since the nefarious Morris worm that appeared in 1988 [139] to demonstrate the potential problems caused due to buggy software and the Internet, the goals of current generation attacks have completely changed. The current attacks, such as worms, viruses, bots, and spyware, are solely motivated towards monetary gains. For example, in 2010 malware cost billion dollars to businesses world-wide [31]. These malware instances get installed on users' machines either by exploiting vulnerabilities present in programs running on systems [15], drive-by-download techniques [88], or enticing the users to download the malicious code inadvertently [123]. Researchers have developed various host-based intrusion detection systems that either employ signatures

or anomalous runtime behaviors to detect malware running on end users' systems.

2.1.1 Signature-based Detection

Signature-based intrusion detection systems use pre-configured and pre-defined attack patterns known as *signatures* to detect attacks. The intrusion detection system (IDS) incorporates this knowledge into a rule set. When a program's code and data is passed to an IDS, it applies rules to determine if any sequence of data matches with any of the rules. If so, it reports that a possible intrusion is underway [76, 81, 129]. For example, anti-virus software uses signatures to detect the presence of malware on users' systems [93]. However, signature-based systems suffer from various problems. Most importantly, they are not able to detect zero day attacks in which signatures are not available before the spread of malware infection [85]. Further, malware authors employ several obfuscation techniques such as polymorphism and metamorphism to defeat these systems [83,89,147]. During obfuscation, malicious code authors generate a new malware sample with the same functionality, but different instruction sequences. With this method, pure syntactic pattern matching solutions fail to match malicious code sequences with the sequences present in signature databases. Mihai et al. [21] presented an approach that addressed the deficiency of syntactic matching solutions by incorporating instruction semantics to detect malicious program traits.

2.1.2 Anomaly-based Detection

Anomaly detectors address the limitations of signature-based intrusion detectors by removing the dependence upon known attacks. These approaches attempt to detect intrusive activities by distinguishing them from normal or benign system activities. For that, an IDS constructs a normal activity profile for users, programs, or systems. Since a normal profile serves as a baseline to detect attacks, extensive research has been done to model users', programs', and systems' normal behavior. Forrest et al. [61] first proposed a system-call based anomaly detector. Their system monitored

all system calls invoked by processes and constructed a benign execution model of programs using system call sequences. With this model, they could detect attacks in which attackers exploited vulnerabilities of programs and subverted their execution. Following the success of this work, many research projects have developed system-call based intrusion detectors [35, 84, 99, 120, 128, 166]. Further, static analysis based approaches have also been developed to automatically derive a model of application behavior [49, 50, 156]. Though anomaly based approaches became popular for the detection of new and unknown attacks, the problem of false positives inhibited their wide spread adoption for commercial purposes [6, 136].

2.2 Event Monitoring

An integral part of the attack detection process is to identify right interfaces and events to be monitored to create precise models of execution of programs. The challenge is to provide balance between richness of the interface and security and performance of the monitor. A security software that monitors programs' execution behaviors must provide complete mediation, tamper-resistance, and verifiability [4, 5]. In this section, we present different monitoring approaches and strategies adopted by previous security systems.

2.2.1 Library Call Monitoring

Usually every commercial program running on systems is made of its own code and the shared library code. The library code provides application developers opportunities to re-use the already developed code and only focus on the implementation of their own logic. In this setting, most of the kernel functions are invoked via library methods. Using this intuition, security researchers have built intrusion detection systems by monitoring the execution of software at the library-call interface [54, 69]. Monitoring this interface offers an advantage: library calls provide much richer information as compared to system calls. Commodity operating systems such as Linux and Windows

have hundreds of system calls, but thousands of library functions. Hence, library call based models can capture fine-grained execution behaviors of programs. Unfortunately, library API monitoring systems do not provide any real security. Knowledgeable attackers can easily bypass library API monitors by directly invoking system calls and perform malicious actions. As mentioned above, a security monitor must provide a complete mediation on operations performed by software that is being monitored.

2.2.2 Kernel Trap Monitoring

To address the drawbacks of library call monitors, researches have created monitors based on the kernel-trap interface [36,39,49,50,61]. Though the kernel-trap or system-call interface is coarser than the library call interface, it offers a complete mediation, a feature missing from library call monitors. The system-call interface is a hardware enforced, non-bypassable interface for user-space processes. Programs, including malware, requesting services from the kernel using this interface would then reveal their malicious behaviors.

Apart from the complete mediation, it is also important to design tamper-resistant monitors. Designing a tamper-resistant monitor requires the knowledge of a trusted computing base (TCB) of a system. As malicious code authors attempt to evade existing security system, they generally attack the security software itself. Due to this reason, a system-call monitor residing in the user-space is the most vulnerable against direct attacks. Given that most systems have operating systems as a part of the TCB, the obvious choice is to deploy the monitor inside the kernel in the form of a driver. However, in recent years, attacks have increased at the kernel-level [135]. Hence, operating systems are no longer a part of the TCB. To secure the monitor, research solutions deploy the monitor inside hypervisors or virtual machine monitors [29,66,96,141]. To record a system-call from the hypervisor, these monitors interpose on the execution of a system call, force it to reach the hypervisor, record

the system call number and its arguments, and resume the execution of the system call inside the operating system. This design choice offered both the protection and complete mediation.

2.2.3 Instruction Monitoring

System-call and library-call monitoring allow security tools to reveal malicious behaviors of programs and detect attacks. The information extracted by these monitors are still coarse-grained. Though the coarse-grained monitoring is preferred for the attack detection due to its low overhead, researchers have also explored ways of extracting fine-grained execution information for other perspectives, such as malware analysis, taint analysis, and information flow control. To extract fine-grained program execution information, instruction-level monitors are developed [29, 101, 169]. These monitors use either single-stepping methods provided by hardware vendors or emulation techniques by running the system inside emulators. Depending upon the trusted computing base, these monitors are also either deployed inside operating systems or hypervisors. Due to its severe overhead, instruction-level monitors are only limited to offline processing and analysis.

2.2.4 VMM Trap Interface Monitoring

So far the monitoring approaches presented here were only focused on applications. With the popularity of virtualization and its use in security, researchers have started looking into ways of monitoring the execution behaviors of operating systems. To this end, they looked at the interfaces present between hypervisors and guest operating systems running inside virtual machines. Hypervisors offer a trap-based interface called *hypercall* to guest operating systems to request services from them, a design similar to the system-call interface presented by operating systems to user-space applications [8]. Srivastava et al. [143] first proposed a hypercall monitoring infrastructure and correlated hypercalls with systems calls to detect new classes of attacks. Later,

hypercall based intrusion detection and prevention systems were developed [80].

With the advent of hardware virtual machines, the trap-based hypercall interface is converted into hardware supported *VMEXIT* instructions [65]. In a recent work, Srivastava et al. [142] introduced a new monitoring architecture that intercepts VMEXIT instructions and correlates them with system calls. Though hypercall based monitoring reveals operating systems' interaction with the hardware, this interface suffers from the semantic gap [17, 30]. For example, a high-level file read operation is visible in the form of disk blocks' read via this interface. Further, the hypercall interface is very noisy due to the frequent hardware accesses performed by operating systems.

This dissertation research utilizes the kernel-trap interface to monitor system calls for the detection of process-to-process parasitic attacks. To withstand attacks launched by kernel-level malware, we deploy our monitor inside the hypervisor. In this way, our monitor achieves both tamper-resistance and the complete mediation. We describe the design of our monitor in Chapter 4.

2.3 Event Correlation

Researchers not only used event monitoring in isolation, but they even tried combining events gathered from different security monitors. The motivation comes from a simple observation that usually enterprises deploy multiple security tools, such as intrusion detection systems, firewalls, antivirus tools, and file integrity checkers, to offer better detection coverage and protect their networks and infrastructure. These security tools may employ different approaches and detect different attacks. For example, anti-virus scanners look for specific signatures to detect attacks, firewalls filter incoming packets into the network, and IDSs detect attacks either by employing traffic signatures or using anomalous behaviors. Though these approaches are capable of detecting specific attacks, they still work in isolation without any co-ordination, and due to this reason

they miss attacks. Researchers noticed this problem and presented correlation-based systems that combine events gathered from multiple tools, process those events, and detect attacks based on the outcome of the correlation [28, 103, 117, 119, 121].

Porras et al. [118] created a distributed IDS design that deploys multiple IDSs in a network, collects the data gathered from these systems, and correlates them to detect attacks. Valdes et al. [153] developed a multi-sensor architecture based on Bayes inference for intrusion detection. They demonstrated the effectiveness of this architecture by improving the sensitivity the IDS and reducing false alarms. They also presented a probabilistic approach for correlating alerts. The probabilistic approach provided a cohesive mathematical framework for the alert correlation. Ning et al. [102] proposed a correlation based approach that constructs attack scenarios using prerequisites and consequences of intrusions. Using prerequisites and consequences of various attacks, their system correlated attacks by matching the consequences of some previous alerts and the prerequisite of some later ones.

Though these systems integrated alerts gathered from multiple sensors, they mostly operated based on responses from network intrusion detection systems. Researchers have also explored solutions that combine both host- and network-level information. Lindqvist et al. [82] presented an intrusion detection system for Solaris operating systems that uses host-level audit logs to complement network-level information. Zeng et al. [170] proposed an approach that considers both the coordination within a botnet and malicious behavior each bot exhibits at the host level. Their system extracted network-level features using the NetFlow data to analyze the similarity or dissimilarity of network behaviors, and on hosts, they monitored behaviors such as a registry creation, file creation, and port opening. Their framework showed the effectiveness in detecting various types of botnets with low false-alarm rates. Gummadi et al. [58] developed a system that differentiates network traffic generated by human from bots. With this design, they reduced spam, denial-of-service traffic,

and click fraud attacks on networks without blocking legitimate traffic. To differentiate between human and bots, they extracted host-level features such keystrokes and mouse movements as these features will not be present in case of malware. Ramachandran [122] proposed a correlation based approach that prevents data leakage in enterprise networks. Their system relies on a host-based trusted component to assist with the tracking of provenance of network traffic, annotate the traffic with taints, and leave the enforcement to devices in the network.

My dissertation research also uses the concept of event correlation in which it combines the host-level software execution information with the network-level packet information to identify malicious processes running on systems. We present the details of our correlation techniques in Chapter 3.

2.4 Operating Systems Security

In previous sections, we presented security solutions and monitoring architectures for user-level applications; we briefly touched upon operating systems' security. In this section, we discuss security of operating systems and show how different kernels have evolved over the years. Though kernel-level malware started compromising systems recently, researchers have thought about them much before and that led to various efforts in designing secure operating systems. These efforts created new kernels and retrofitted security into the commodity kernels.

2.4.1 Multics

The Multics project, a harbinger of secure operating systems, started in 1965 and evolved as a commercial operating system in 1973. The goal of the Multics project was to build a general purpose, time sharing operating system that could offer both security and performance to its users. Its protection model consists of three elements: access control, rings and brackets, and multilevel security. For the access control, each object contains an access control list, detailing what operations could be performed by

which process on that particular object. Its ring and bracket based protection further limited the access and provided fine-grained protection. Each segment is related to a ring that details read, write, and execute operations of processes over that segment. Further, brackets define the transition rules among rings. Finally, Multics pioneered the design and implementation of *Multilevel Security*, which prohibits a process from reading data that is more secret than the process itself or writing data to less secret objects. The shortcomings of the Multics project included its ambitious goal to become a general purpose operating system with the great security and performance. As realized by researchers at that time, it was not possible to have all these goals satisfied together given that state of the art hardware in 70's. Though the Multics project was not a great success, it brought many new ideas that set the foundation for the design of future secure operating systems.

2.4.2 Security Kernels

As the Multics project was winding down, researchers and industry professionals started realizing the need of secure operating systems. Two directions emerged: the first direction aimed to design operating systems that achieve generality and performance with the limited security, and the second direction moved towards secure and verifiable operating systems with reasonable performance for a limited set of applications. The second direction led us to security kernels [4]. The idea of a security kernel was to build a small kernel that provides both verifiability and performance. Since the TCB was reduced, it became easy to verify the properties of the kernel. Further, the security kernel was customized to remove performance bottlenecks. This design attempted to remove both shortcomings of Multics – verifiability and performance. With this design in mind, between 1970 and 1980, many commercial projects started to build secure operating systems from scratch. To name a few, Secure Communication Processor (Scomp) [40] from Honeywell, the Gemini Secure Operating System

(GSOS) [134] from Gemini, LOCK systems from Secure Computing [127], and Kernelized Secure Operating System (KSOS) [95] from Ford Aerospace and Communications. Though security kernels brought verifiability in systems due to their minimal trusted computing base, they were far from perfect. Security kernels rely on trusted subjects or processes. These trusted processes were not the part of the TCB, but had to be trusted to perform critical tasks outside the security kernel. The claim was made that these trusted subjects must adhere to the same engineering principles as the security kernel to achieve verifiability, however, in practice ensuring the correct behavior of trusted services was not easy as general purpose systems would have many trusted processes.

2.4.3 Separation Kernels

Addressing the problems of security kernels, Rushby designed a new kernel called *separation kernel* [125]. Fundamental to the separation kernel is the independence and authorized communications among trusted services. In separation kernels, each trusted service executes inside an isolated and independent system. These isolated execution containers may also execute on the same physical hardware. This separation among trusted services allows the kernel to interpose on all communications among them and keeps them isolated throughout the execution. Later, Rushby formally proved the separation guarantees provided by the kernel [126]. Though this design looks similar to virtualized environments where each operating system runs in its own virtual machine, the key difference between separation kernels and virtualization is that the separation kernel does not require hardware to be virtualized – a key requirement for virtualized systems. The Multiple Independent Levels of Security (MILS) is an example of a separation kernel based system [59].

2.4.4 Micro Kernels

Microkernel systems appeared in 80s, and they were similar to security kernels in their design of trusted computing base. The goal of a microkernel was to offer same security guarantees as security kernels, but provide ease of development and flexibility to developers. The Mach microkernel [1] consisted of a small security kernel and supported complete operating system constructions, including message passing between different subsystems (IPC), memory management, and scheduling. The small kernel design was achieved by moving out other code such as drivers, file systems, and servers into user-space. This design improved the reliability and security of the system because most of these problems are caused by drivers. Moving drivers out from the kernel reduced these problems. Microkernel designs were further used to create high-assurance systems such as KeyKOS and EROS. However, the extensive use of IPC to communicate among different components impacted systems' performance severely [60].

This dissertation utilizes the concept of driver isolation directly. Security problems caused by kernel-level malware and untrusted drivers are growing. As described in Chapter 5, we present software that isolates drivers of commodity operating systems in a separate address space and monitors their interaction with the kernel code.

2.4.5 Commodity Monolithic Kernels

Multics project triggered the design of secure operating systems and also motivated researchers to aim in the direction of operating systems that achieve generality and performance with the limited security. These efforts created the UNIX operating system, and later, Linus reimplemented the POSIX specification to create a completely free Linux operating system [68]. UNIX and Linux are a monolithic kernel based operating systems. In this design, the TCB is not small as it contains the core kernel along with all drivers. Due to this monolithic design, commodity kernels offer better

performance as compared to microkernels. However, their security is poor, and they are never considered to be used for high assurance purposes.

The primary security flaw comes from kernel drivers as they reside in the kernel. Buggy drivers can easily crash the whole system. Further, kernel malware instances install themselves in the form of drivers, and due to a unified address space malware tampers with the kernel code and data. Many security projects attempted to retrofit commodity kernels with protections by adopting designs similar to microkernels. Nooks [145] isolated drivers using paging hardware in a separate address from the kernel to improve the reliability of the system. Though the approach demonstrated its usefulness, it suffered from performance problems. Ganapathy et al. [44] split drivers between user and kernel space components and moved non-security critical code into user-space to improve the reliability of the system. In further works, they also retrofitted authorization policies and security hooks in commodity kernels [42,43].

This thesis also attempts to retrofit security into commodity Linux and Windows operating systems. We protect kernel code and data against kernel-level rootkits and parasitic attacks. Our solution utilizes memory page protection and runtime monitoring techniques to achieve security. To achieve performance goals, we use object partitioning, binary rewriting, and code generation.

2.5 Virtualization Security

Virtualization supports the execution of multiple operating systems on one computer using a piece of software called a virtual machine monitor (VMM). The environment created by the VMM is called a virtual machine [116]. Popek and Goldberg first laid out the requirements for systems supporting virtualization [116]. They identified three essential features to be supported by a VMM: efficiency, mediation, and identical behavior. A VMM must not severely impact the performance of software

executing inside a virtual machine. In other words, the majority of a virtual processor's instructions should be executed directly by the real processor, with no software intervention by the VMM. A VMM must be completely in control of the systems' resources all the time. A rogue VM must not use resources that have not been assigned to it. Finally, the identical behavior requirement states that a program should exhibit the same behavior as it would have behaved in the real environment. Based on these requirements, KVM/370 system was designed. KVM/370 retrofitted the security into the existing IBM VM/370 [53]. Since KVM/370 systems had to use the existing codebase of VM/370, the performance of the overall system was poor. This motivated the design of a VAX VMM security kernel in 1981. This was the first project that designed a virtualized architecture from scratch [72]. Though the VAX VMM's design solved problems present in KVM/370, still it did not see widespread deployment and project was cancelled in 1990. Though the exact reasons are not known, some of them are described by Krager et al. [72].

VMware in 2000 revived the industry by providing virtualized solutions for ubiquitous x86 processors [2, 155]. They adopted a dynamic translation based approach to trap on privileged operations performed by guest operating systems to offer complete mediation. Further, with the advent of hardware support for virtualization [65], researchers and industry started looking into virtualized architectures actively. At this time, security researchers have also started exploring new designs for placing security software. This was required due to a rise in kernel-level malware and increased attacks on operating systems.

On looking closely into virtualized environments, three properties provided by modern virtualized solutions appeared to be well suited for security needs. The first property is *isolation* that states programs running inside one virtual machine cannot be accessed or modified by programs from other virtual machines. This property is attractive for deploying security tools as malicious software cannot attack security

software running inside a virtual machine even if attackers completely control the monitored virtual machine. The second property is the *inspection* that allows VMMs to have complete access to the entire state of guest operating systems running inside virtual machines. In this way, it is difficult for malicious software inside VMs to evade virtual machine based intrusion detection systems. Finally, the last property is *interposition*, which allows a VMM to trap guests on privileged instructions and provides the ability to change a guest VM's execution path.

Researchers have built systems utilizing all three properties provided by virtualized environments. Garfinkel et al. [46] invented a virtual machine introspection based architecture for intrusion detection systems by utilizing the isolation and inspection properties of VMMs. Virtual machine introspection (VMI) is the process of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it. They implemented their prototype system for the VMware hypervisor [155] along with a suite of simple intrusion detection policies to detect attacks running inside the monitored VM. The open source XenAccess project [164] also used a similar approach and developed introspection tools for memory and disks [112] using the Xen virtual machine monitor [8]. Jones et al. [70] used interposition properties to develop a mechanism for virtualized environments that tracks kernel operations related to specific process actions. With their system, process creation, context-switch, and destruction can be observed from a VMM. Further, they [71] used their system to find processes hidden in the kernel memory.

The solutions described in this thesis are also created for virtualized environments. Our approaches utilize isolation to design tamper resistant software by deploying in trusted virtual machines and hypervisors. The inspection and interposition techniques are used for process attribution and the protection of kernel code and data. We present the description of other virtualized security systems in Chapters 3, 4, 5, and 6.

CHAPTER III

PROCESS ATTRIBUTION

3.1 Motivation

As described in Chapter 1, a common feature of different classes of malware such as bots, spyware, worms, and backdoors is their interest in the network for botnet, spam, and denial of service attacks [144]. After detecting a suspicious network flow, network intrusion detection systems (NIDSs) can pinpoint a host within a network or enterprise responsible for that traffic [56, 111]. Such network-level security software can identify an infected system's IP address, network ports, and traffic behaviors. However, they fail to identify individual host-level processes or malicious code responsible for malicious traffic. The coarse-grained information visible to network-level security tools only provides coarse-grained attribution of malicious behaviors: a complete system is considered malicious instead of a process responsible for the malicious activity. This coarse-grained attribution also permits only coarse-grained responses: an administrator could excise an infected system from the network, possibly for re-imaging.

Host-level security tools have full visibility to identify malicious processes or code performing illegitimate activities, including sending or receiving suspicious traffic. However, malware instances can directly affect an application-level security software in execution. The architecture of these malware instances frequently combines a user-level application performing network activity with a kernel-level module that hides the application from the view of host-level security software. The malicious application, likely running with full system privileges, may halt the execution of the security software. Similarly, the malicious kernel component may alter the hooks used by an in-kernel module supporting the user-level security tools so that the security

related checks are simply never invoked as data passes to and from the network. Conventional application-level security tools fail under these direct attacks.

Our goal is to develop software that withstands direct attacks from malware at the application or the kernel layer and provides identification of malicious processes running in the infected system. In this chapter, we leverage the benefits of both host-level information and virtual machine isolation to develop *a tamper-resistant process attribution software that identifies a malicious process that is the end-point of malicious connections.* Such software needs good visibility of the monitored system so that it can correlate network flows with processes, but it also needs strong isolation from any user-level or kernel-level malware that may be present. We architect a process attribution software resistant to direct attacks from malicious software on the infected system. As shown in Figure 1, our design isolates the attribution software in a trusted virtual machine (VM) and relies on the hypervisor to limit the attack surface between any untrusted VM running malware and the trusted VM. Our attribution software, executing in the trusted VM, gets complete visibility of the untrusted VM by using virtual machine introspection (VMI) [46] to identify the process in another VM that is connected to a suspicious network flow.

3.2 Previous Approaches

Prior research has contributed to the development of conventional host-based security tools. Mogul et al. [97] developed a kernel-resident packet filter for UNIX that gave user processes flexibility in selecting legitimate packets. Venema [154] designed a utility to monitor and control incoming network traffic. These traditional software performed filtering based on restrictions inherent in network topology and assumed that all parties inside the network were trusted. As part of the security architecture of the computer system, they resided in kernel-space and user-space, and hence were vulnerable to direct attack by malicious software.

The recent support for virtual machines by commodity hardware has driven development of new security services deployed with the assistance of VMs [45, 114, 159]. Garfinkel et al. [47] showed the feasibility of implementing distributed network-level firewalls using virtual machines. In another work [46], they proposed an intrusion detection system design using virtual machine introspection of an untrusted VM. Our system applies virtual machine introspection to a different problem, using it to correlate network flows with the local processes bound to those flows. Other research used virtual machines for malware detection. Borders et al. [11] designed a system, Siren, that detected malware running within a virtual machine. Yin et al. [169] proposed a system to detect and analyze privacy-breaching malware using taint analysis. Jiang et al. [67] presented an out-of-the-box VMM-based malware detection system. Their proposed technique constructed the internal semantic views of a VM from an external vantage point. In another work [66], they proposed a monitoring tool that observes a virtual machine based honeypot's internal state from outside the honeypot. As a pleasant side-effect of malicious network flow detection and process correlation, our software can often identify processes in the untrusted VM that comprise portions of an attack.

Previous research has developed protection strategies for different types of hardware-level resources in the virtualized environment. Xu et al. [167] proposed a VMM-based usage control model to protect the integrity of kernel memory. Ta-Min et al. [148] proposed a hypervisor based system that allowed applications to partition their system call interface into trusted and untrusted components. Our software, in contrast, protects network resources from attack by malware that runs inside the untrusted virtual machine by monitoring the illegitimate network connections attempts.

These previous hypervisor-based security applications generally take either a network-centric or host-centric view. Our work tries to correlate activity at both levels. It monitors network connections but additionally peers into the state of the running,

untrusted operating system to make its judgments about each connection's validity. Moreover, our software easily scales to collections of virtual machines on a single physical host. A single instance of of the software can also act as an application-level firewall for an entire network of VMs as explained in Section 3.7.

3.3 Overview

We begin with preliminaries. Section 3.3.1 explains our threat model, which assumes that attackers have the ability to execute the real-world attacks infecting widespread computer systems today. Section 3.3.2 provides a brief overview of Xen-based virtual machine architectures and methods allowing inspection of a running VM's state.

3.3.1 Threat Model

We assume that attackers have abilities commonly displayed by real-world attacks against commodity computer systems. Attackers can gain superuser privilege from remote. Attackers are external and have no physical access to the attacked computers, but they may install malicious software on a victim system by exploiting a software vulnerability in an application or operating system or by enticing unsuspecting users to install the malware themselves. The software exploit or the user often executes with full system privileges, so the malware may perform administrative actions such as kernel module or driver installation. Hence, malicious code may execute at both user and kernel levels.

Our system has requirements for correct execution. As with all requirements, an attacker who is able to violate any requirement is likely able to escape detection. Our two requirements of note center on basic expectations for the in-memory data structures used by the kernel that may be infected by an attack.

First, we expect to be able to find the head of linked data structures, often by extracting a kernel symbol value at boot time. An attacker could conceivably cause our attribution software to inspect the incorrect kernel information by replicating

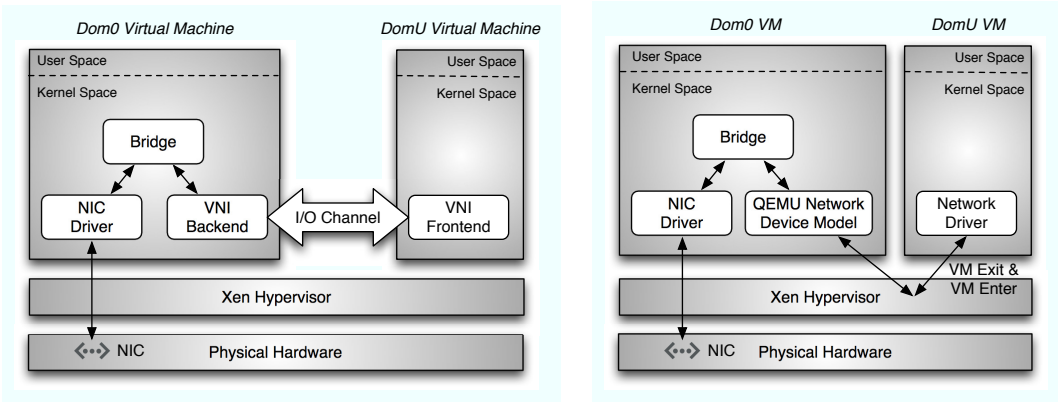
the data structure elsewhere in kernel memory and by altering all code references to the original structure to instead refer to the new structure. Our software would then analyze stale data. It is not immediately clear that such an attack is plausible; moreover, our tool could periodically verify that code references to the data match the symbol value extracted at boot.

Second, we expect that attacks do not alter the ordering or length of fields in aggregate data structures. Our attribution software is preprogrammed with type information about kernel structures, and an attack that alters the structure types would cause our system to read incorrect information from kernel memory. Successfully executing this attack without kernel recompilation appears to be complex, as all kernel code that accesses structure fields would need to be altered to use the attacker's structure layout. As a result, we believe that relying upon known structure definitions is not a limiting factor to our design.

3.3.2 Virtual Machine Introspection

Our design makes use of virtual machine technology to provide isolation between malicious code and our security software. We use Xen [8], an open source hypervisor that runs directly on the physical hardware of a computer. The virtual machines running atop Xen are of two types: unprivileged domains, called domU or guest domains, and a single fully-privileged domain, called dom0. We run normal, possibly vulnerable software in domU and deploy our process attribution software in the isolated dom0.

Xen virtualizes the network input and output of the system. Dom0 is the device driver domain that runs the native network interface card driver software. Unprivileged virtual machines cannot directly access the physical network card, so Xen provides them with a virtualized network interface (VNI). The driver domain receives all the incoming and outgoing packets for all domU VMs executing on the physical system. Dom0 provides an Ethernet bridge connecting the physical network card



Paravirtualized domain Fully virtualized domain

Figure 2: Xen networking architecture.

to all virtual network devices provided by Xen to the domU VMs. (Xen offers other networking modes, such as network address translation, that are not used in our work and will not be considered further.) Dom0 uses its virtual bridge to multiplex and demultiplex packets between the physical network interface and each unprivileged virtual machine's VNI. Figure 2 shows the Xen networking architecture when the virtual machines' network interfaces are connected through a virtual Ethernet bridge. The guest VMs send and receive packets via either an I/O channel to dom0 or emulated virtual devices.

The strong isolation provided by a hypervisor between dom0 and the guest domains complicates the ability to correlate network flows with software executing in a guest domain. Yet, dom0 has complete access to the entire state of the guest operating systems running in untrusted virtual machines. *Virtual machine introspection* (VMI) [46] is a technique by which dom0 can determine execution properties of guest VMs by monitoring their runtime state, generally through direct memory inspection. VMI allows security software to remain protected from direct attack by malicious software executing in a guest VM while still able to observe critical system state.

Xen offers low-level APIs to allow dom0 to map arbitrary memory pages of domU

as shared memory. XenAccess [164] is a dom0 userspace introspection library developed for Xen that builds onto the low-level functionality provided by Xen. The attribution software uses XenAccess APIs to map raw memory pages of domU's kernel inside dom0. It then builds higher-level memory abstractions, such as aggregate structures and linked data types, from the contents of raw memory pages by using the known coding semantics of the guest operating system's kernel. Our attribution software inspects these meaningful, higher-level abstractions to determine how applications executing in the guest VM use network resources.

3.4 Network-to-Host Correlation

The process attribution software is designed to resist the direct attacks possible in our threat model. The architecture of the software is driven by the following three goals:

- **Tamper Resistance:** The attribution software should continue to function reliably and verify all network connections even if an attacker gains entry into the monitored system. In particular, the design should not rely on components installed in the monitored host as processes or kernel modules, as these have been points of direct attack in previous security tools.

- **Independence:** It should work without any cooperation from the monitored system. In fact, the system may not be aware of the presence of the attribution software.

- **Lightweight Verification:** Our intent is to use the process attribution software for online verification of network connections to real systems. The design should allow for efficient monitoring of network traffic and correlation to applications sending and receiving that traffic.

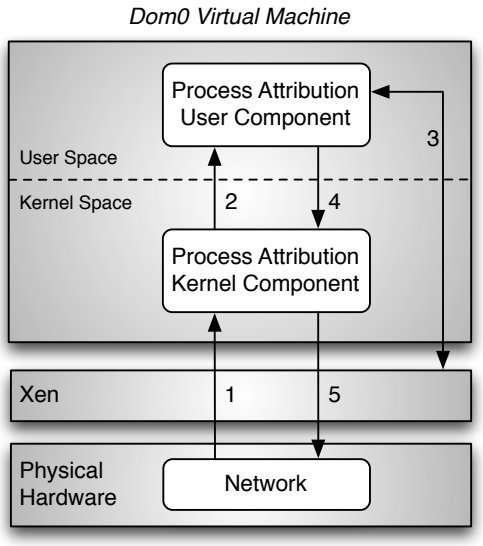


Figure 3: The high-level architecture of the process attribution software. (1) Packets inbound to and outbound from a guest domain are processed by dom0. (2) The kernel component intercepts the packets and passes them to a user-space component for analysis. (3) The user-space component uses virtual machine introspection to identify software in a guest domain processing the data. (4) The user-space component sends a response back to the kernel component. (5) Packets will be placed on the network.

Our software design, shown in Figure 3, satisfies these goals by leveraging virtual machine isolation and virtual machine introspection. Its entire software runs within the privileged dom0 VM, and it hooks into Xen's virtual network interface to collect and filter all guest domains' network packets. Since the hypervisor provides strong isolation among the virtual machines, this design achieves the first goal of tamper-resistance.

In order to provide process attribution, our software must identify the process that is sending or receiving packets inside domU. The sensor correlates packet and process information by directly inspecting the domU virtual machine's memory via virtual machine introspection. It looks into the kernel's memory and traverses the data structures to map process and network information. This achieves our second design goal of independence, as there are no components of the sensor inside domU. Our introspection procedure rapidly analyzes the kernel's data structures, satisfying

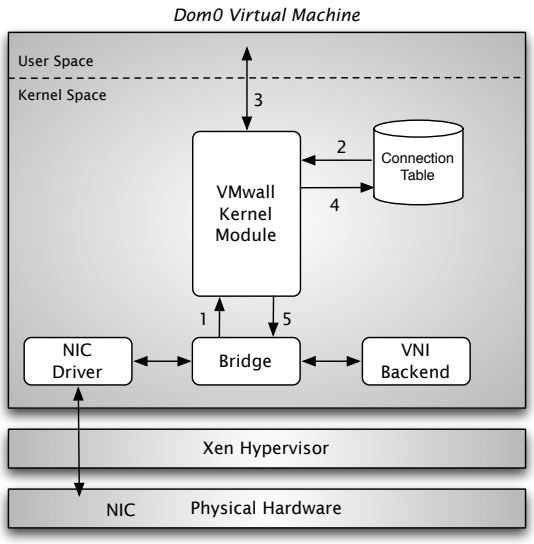


Figure 4: The kernel module architecture of the process attribution software. (1) Packets inbound to and outbound from a guest domain are intercepted and passed to the kernel module. (2) The module receives each packet and looks into its connection table to decide if an introspection request is to be made. (3) It sends an introspection request to the user agent and receives the response back. (4) On the successful introspection, the kernel module adds the result to process future packets from the same connection. (5) The kernel module allows the packets to go through.

the third goal of lightweight verification.

The high-level design of the our software has two components: a kernel module and user agent, both in dom0. The kernel component inspects packets and identifies separate connections. The user agent performs introspection to extract information about processes executing in guest VMs. Sections 3.4.1 and 3.4.2 present detailed information about the two components.

3.4.1 Kernel Component

The kernel component is a module loaded inside the dom0 Linux kernel. Figure 4 presents the kernel module's complete architecture and steps involved in processing the packet inside the kernel. The kernel component intercepts all network packets to or from untrusted virtual machines. Interception occurs by hooking into Xen's network bridge between the physical interface card and virtual network interface.

When the kernel component intercepts a packet, it checks a connection table to see if packets for this connection has already been attributed to a host-level process inside the untrusted VM. If this is a new connection, the kernel component identifies separate traffic flows. Whenever it receives the first packet for a connection, it extracts the source and destination IP addresses and ports, which it then passes to the userspace component for further use. The kernel component is a passive network tap and allows all packets flows to continue unimpeded. The user agent performs introspection and sends the result of introspection back to the kernel module. The introspection result shows whether a process bound the connection is identified successfully or not, and the kernel component stores this information in the connection table. On the successful introspection, further packets from the same connection are processed without performing introspection.

3.4.2 User Agent

The user agent uses virtual machine introspection to correlate network packets and processes. It receives introspection requests from the kernel component containing network information such as source port, source IP address, destination port, destination IP address, and protocol. It first uses the packet's source (or destination) IP address to identify the VM that is sending (or receiving) the packet. When it finds the VM, it then tries to find the process that is bound to the source (or destination) port.

The user agent maps a network port to the domU process that is bound to the port. As needed, it maps domU kernel data structures into dom0 memory. Process and network information is likely not available in a single data structure but instead is scattered over many data structures. The user agent works in steps by first identifying the domU kernel data structures that store IP address and port information. Then, the user agent identifies the process handling this network connection by iterating

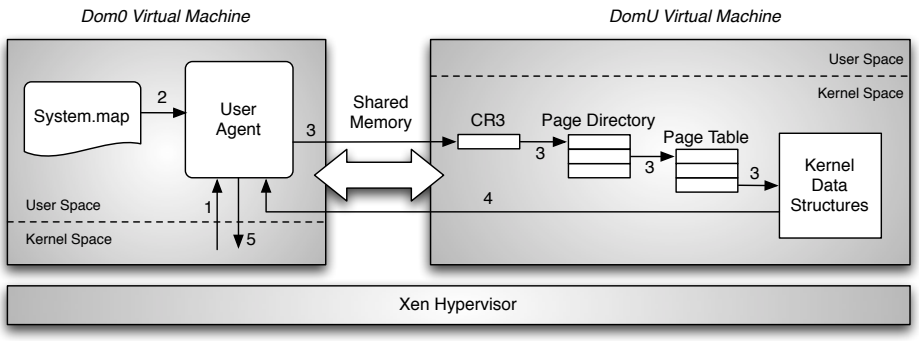


Figure 5: The user agent's architecture. (1) The user agent receives the introspection request. (2) The user agent reads the symbol file to extract the kernel virtual addresses corresponding to known kernel symbols. (3) The user agent uses Xen to map the domU kernel memory pages containing process and network data structures. (4) It traverses the data structures to correlate network and process activity. (5) The user agent sends a response to the kernel module.

over the list of running processes and checking each process to see if it is bound to the port. When it finds the process bound to the port, it extracts the process' identifier, its name, and the full path to its executable. Figure 5 shows the steps involved in mapping a domU memory page, containing kernel data inside the dom0 address space using kernel symbols.

3.5 Implementation

We have implemented a prototype of the process attribution software using the Xen hypervisor for both Linux and Windows guest operating systems. Its implementation consists of two parts corresponding to the two pieces described in the previous section: the kernel module and the user agent. The following sections describe specific details affecting implementation of the two architectural components.

3.5.1 Extending ebtables

Our kernel module uses a modified ebtables packet filter to intercept all packets sent to or from a guest domain. Ebtables [22] is an open source utility that filters packets at an Ethernet bridge. Whenever ebtables accepts packets based on its coarse-grained

rules, we hook the operation and invoke kernel module for our additional checks. We modified ebtables to implement this hook, which passes a reference of the packet to the kernel component.

3.5.2 Accessing DomU Kernel Memory

The user agent uses the XenAccess introspection library [164] to access domU kernel memory from dom0. It maps domU memory pages containing kernel data structures into the virtual memory space of the user agent, executing in the trusted VM. XenAccess provides APIs that map domU kernel memory pages identified either by explicit kernel virtual addresses or by exported kernel symbols. In Linux, the exported symbols are stored in the file named System.map, while in Windows exported symbols can be extracted from the debug information. The user agent utilizes certain domU data structures that are exported by the kernel and hence mapped with the help of kernel symbols. Other data structures reachable by pointers from the known structures are mapped using kernel virtual addresses. The domU virtual machine presented in Figure 5 shows the internal mechanism involved to map the memory page that contains the desired kernel data structure.

3.5.3 Parsing Kernel Data Structures

To identify processes using the network, the user agent must be able to parse high-level kernel data structures from the raw memory pages provided by XenAccess. Extracting kernel data structures from the mapped memory pages is a non-trivial task. For example, Linux maintains a doubly-linked list that stores the kernel's private data for all running processes. The head pointer of this list is stored in the exported kernel symbol `init_task`. If we want to extract the list of processes running inside domU, we can map the memory page of domU that contains the `init_task` symbol. However, the agent must traverse the complete linked list and hence requires the offset to the `next` member in the process structure. We extract this information

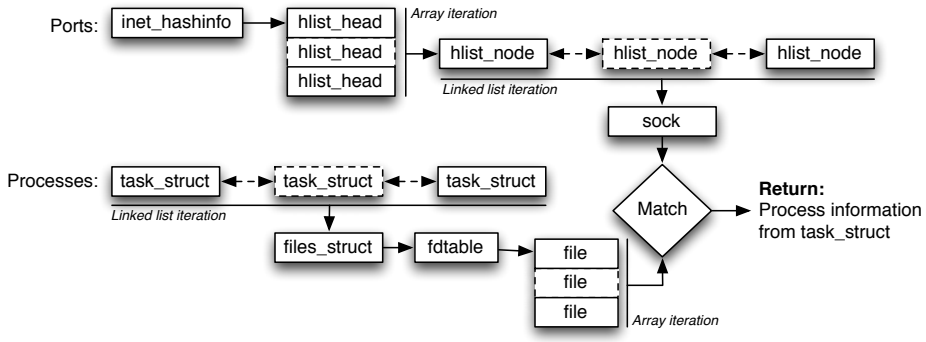


Figure 6: DomU Linux kernel data structures traversed by the user agent during correlation of the process and TCP packet information.

offline directly from the kernel source code and use these values in the user agent. This source code inspection is not the optimal way to identify offsets because the offset values often change with the kernel versions. However, there are other automatic ways to extract this information from the kernel binary if it was compiled with a debug option [87].

This provides the agent with sufficient information to traverse kernel data structures. It uses known field offsets to extract the virtual addresses of pointer field members from the mapped memory pages. It then maps domU memory pages by specifying the extracted virtual addresses. This process is performed recursively until the agent traverses the data structures necessary to extract the process name corresponding to the source or destination port of a network communication. Figure 6 shows the list of the kernel data structures traversed by the user agent to correlate a TCP packet and process information. First, it tries to obtain a reference to the socket bound to the port number specified in the packet. After acquiring this reference, it iterates over the list of processes to find the process owning the socket.

Identifying a process corresponding to a network connection is much harder for Windows operating systems due to the unavailability of the source code. To achieve our goals, we have reverse engineered part of the Windows kernel to identify these structures that store the correlation information. Unfortunately, Windows does not

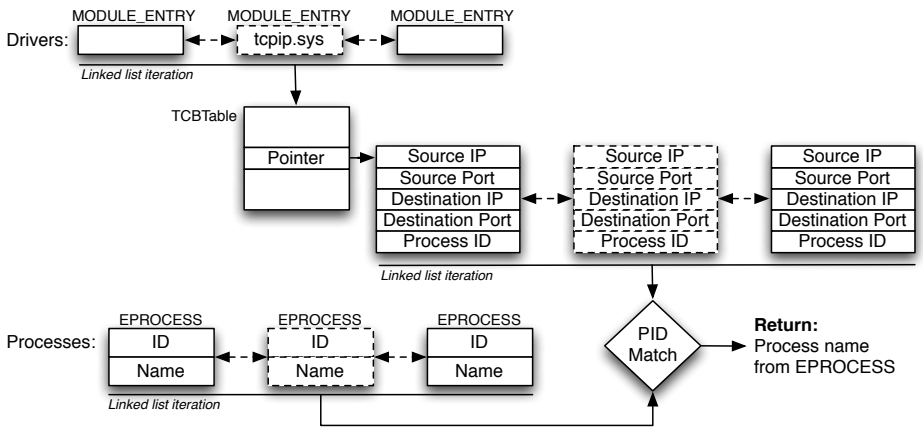


Figure 7: Network connection to host-level process correlation in Windows

store network port and process name information in a single structure. A network driver (`tcpip.sys`) manages network connection related information. To locate the data structure corresponding to `tcpip.sys`, the userspace component iterates across the kernel's list of loaded drivers to find the structure's memory address. The driver maintains a pointer to a structure called `TCBTable`, which in turn points to a linked list of objects containing network ports and process IDs for open connections. To convert the process ID to a process name, the component iterates across the guest kernel's linked list of running processes. Figure 7 illustrates the complete process of resolving a network connection to a host-level process name.

3.6 Evaluation

The basic requirement for the process attribution software is to identify a process that is the end-point of a connection. Our software must be able to perform the correlation irrespective of whether the process is benign or malicious. We tested the attribution software against both Linux and Windows based backdoors, worms, and bots that attempt to use the network for malicious activity. Section 3.6.1 tests the attribution software against attacks that receive inbound connections from attackers or connect out to remote systems. Section 3.6.2 tests legitimate software in the presence of the

attribution software.

3.6.1 Illegitimate Connections

We first tested attacks that receive inbound connections from remote attackers. These attacks are rootkits that install backdoor programs. The backdoors run as user processes, listen for connections on a port known to the attacker, and receive and execute requests sent by the attacker. We used the following backdoors:

- **Blackhole** runs a TCP server on port 12345 [105].
- **Gummo** runs a TCP server at port 31337 [105].
- **Bdoor** runs a backdoor daemon on port 8080 [105].
- **Ovas0n** runs a TCP server on port 29369 [105].
- **Cheetah** runs a TCP server at the attacker's specified port number [105].

Once installed on a vulnerable system, attacks such as worms and bots may attempt to make outbound connections without getting prompted from a remote attacker. We tested our software with the following pieces of malware that generate outbound traffic:

- **Apache-ssl** is a variant of the Slapper worm that self-propagates by opening TCP connections for port scanning [106].
- **Apache-linux** is a worm that exploits vulnerable Apache servers and spawns a shell on port 30464 [106].
- **BackDoor-Rev.b** is a tool that is be used by a worm to make network connections to arbitrary Internet addresses and ports [92].
- **Q8** is an IRC-based bot that opens TCP connections to contact an IRC server to receive commands from the botmaster [62].

Table 1: List of legitimate software with which the process attribution software is tested.

Name	*Connection Type*
rcp	Outbound
rsh	Outbound
yum	Outbound
rlogin	Outbound
ssh	Outbound
scp	Outbound
wget	Outbound
tcp client	Outbound
putty	Outbound
Internet Explorer	Outbound
thttpd	Inbound
tcp server	Inbound
sshd	Inbound

- **Kaiten** is a bot that opens TCP connections to contact an IRC server [108].
- **Coromputer Dunno** is an IRC-based bot, providing basic functionalities such as port scanning [55].
- **AdClicker.BA** is a trojan that makes malicious connections.

The process attribution software successfully resolved all connections attempted by malware instances. In all cases, both sending and receiving, the kernel component intercepted the first packet of each connection and passed it to the userspace component. The userspace component successfully found the process that is the end-point of the connection and informed the kernel component.

3.6.2 Legitimate Connections

We also evaluated the ability of our software to resolve legitimate connections made by processes running inside domU. We selected a few network applications both for Windows and Linux. We then ran these applications inside domU. Table 1 shows the list of processes that we tested and the type of connections used by the processes.

Our software allowed all connections made by these applications. The `yum` application, a package manager for Fedora Core Linux, had runtime behavior of interest. In our test, we updated domU with the `yum update` command. During the package update, `yum` created many child processes with the same name `yum`, and these child processes made network connections. The attribution software successfully validated all the connections via introspection and allowed their network connections.

3.6.3 Performance Evaluation

The process attribution software verifying all packets traversing a network may impact the performance of applications relying on a timely delivery of those packets. We investigated the performance impact of our software as perceived by network applications running inside the untrusted virtual machine. We performed experiments both with and without the attribution software running inside dom0. All Linux-based experiments were conducted on a machine with an Intel Core 2 Duo T7500 processor at 2.20 GHz with 2 GB RAM. Both dom0 and domU virtual machines ran 32 bit Fedora Core 5 Linux. DomU had 512 MB of physical memory, and dom0 had the remaining 1.5 GB. The versions of Xen and XenAccess were 3.0.4 and 0.3, respectively. We performed our experiments using both TCP and UDP connections. All reported results show the median time taken from five measurements. We measured microbenchmarks with the Linux `gettimeofday` system call and longer executions with the `time` command-line utility.

Our Windows based experiments were carried out on an Intel Core 2 Quad 2.66 GHz system. We assigned 1 GB of memory to the untrusted Windows XP SP2 VM and 3 GB combined to the Xen hypervisor and the high-privilege Fedora Core 9 VM. We measured networking overheads using `IBM Page Detailer` [64] and `wget`. We executed all measurements five times and present here the median values.

The attribution software's performance depends on the introspection time taken

Table 2: Introspection time (μs) taken by the process attribution software to perform correlation of network flow with the process executing inside domU.

Configuration	*TCP Introspection Time*	*UDP Introspection Time*
Inbound Connection to domU	251	438
Outbound Connection from domU	1080	445

by the user component. We measured the introspection time for the Linux guest OS both for incoming and outgoing connections to and from domU. Table 2 shows the results of experiments measuring introspection time. It is evident that the introspection time for incoming TCP connections is very small. Strangely, the introspection time for outgoing TCP connections is notably higher. The reason for this difference lies in the way that the Linux kernel stores information for TCP connections. It maintains TCP connection information for listening and established connections in two different tables. TCP sockets in a listening state reside in a table of size 32, whereas the established sockets are stored in a table of size 65536. Since the newly established TCP sockets can be placed at any index inside the table, the introspection routine that iterates on this table from dom0 must search half of the table on average.

We also measured the introspection time for UDP data streams. Table 2 shows the result for UDP inbound and outbound packets. In this case, the introspection time for inbound and outbound data varies little. The Linux kernel keeps the information for UDP streams in a single table of size 128, which is why the introspection time is similar in both cases.

Next, we measured the attribution software' performance on the Windows guest operating system during network operations. Using the *IBM Page Detailer*, we measured the time to load a complex webpage (`http://www.cnn.com`) that consisted of many objects spread across multiple servers. The page load caused the browser to make numerous network connections—an important test because the kernel component intercepts each packet and performs introspection on SYN packets. The result,

Table 3: Results of the network performance tests for unmonitored execution and for the process attribution software's monitoring; smaller measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Process Attribution Software*	%
Page Loading (sec)	3.64	3.82	4.95
Network File Copy (sec)	38.00	39.00	2.63

shown in Table 3, demonstrates that the overhead of the attribution software is low.

We next executed a network file transfer by hosting a 174 MB file on a local networked server running `thttpd` and then downloading the file over HTTP using `wget` from the untrusted VM. Table 3 shows that our software incurred less than 3% overhead on the network transfer; we expect that this strong performance is possible because its packet interception design does not require it to queue and delay packets.

3.6.4 Security Analysis

The process attribution software relies on particular data structures maintained by the domU kernel. An attacker who fully controls domU could violate the integrity of these data structures in an attempt to bypass the introspection. To counter such attacks, we present the data structure protection software in Chapter 6.

Attackers can also try to cloak their malware by appearing to be benign software. They can then rename their malicious binary to the name of a benign process. Our software counters this problem by extracting the full path to the process on the guest machine. Attackers could then replace the complete program binary with a trojaned version to evade the full path verification. The attribution software itself has no defenses against this attack, but previous research has already addressed this problem with disk monitoring utilities that protect critical files [23, 112].

In the current design, the process attribution software performs introspection once per connection. This design incurs low overhead on network applications running inside the untrusted VM as demonstrated by our experiments. An attacker may use this

design to send malicious packets without invoking the process attribution software's user agent. An attacker could hijack a connection after it has been intercepted and introspected by our software. Then, he can use the already established connection to send attack packets. Since these packets are a part of already established connections, there will be no introspection request for them. One approach to counter certain instances of connection hijacking attacks is to perform introspection for randomly chosen packets to verify whether the packets belong to the same process or not. Detection of subtle hijacking attempts may require deep packet inspection capability in the process attribution software that would increase the overhead.

Finally, attackers could hijack a process by exploiting a vulnerability, and they could then change its in-memory image. We term this behavior as a parasitic behavior. To address this problem, we propose software in Chapter 4 to detect process manipulation attacks.

3.7 Other Application: VMwall

We have described the process attribution software that correlates network packets to host-level processes to identify malware running inside untrusted virtual machines, sending or receiving malicious traffic. In the design explained so far, the attribution software does not decide whether the traffic is to be blocked or denied. The legitimacy of the traffic is decided by network intrusion detection systems as shown in Figure 1. However, an ability of the process attribution software to determine a process sending and receiving traffic is legitimate or not creates an avenue to design a new tamper-resistant application-aware firewall that can provide the combined features of both host- and network-level firewall. To this end, we create a new virtual machine based firewall called *VMwall* by extending the design of process attribution software. As with the process attribution software, VMwall operates from the trusted virtual machine and uses VMI to find the end-point of the connection. In addition to

this, VMwall maintains a pre-defined security policy to decide whether a connection should be allowed or blocked.

3.7.1 Policy Design and Rules

VMwall identifies legitimate connections via a whitelist-based policy listing processes allowed to send or receive data. Each process that wants to communicate over the network must be specified in the whitelist *a priori*. This whitelist resides inside dom0 and can only be updated by administrators in a manner similar to traditional application-level firewalls. The whitelist based design of VMwall introduces some usability issues because all applications that should be allowed to make network connections must be specified in the list. This limitation is not specific to VMwall and is inherent to the whitelist based products and solutions [16, 48].

3.7.2 Modifications to the Kernel Component

In its current design, the kernel component does not block any traffic. To be able to act as a firewall, the kernel component must be able to block the traffic if the corresponding process is not specified in the whitelist. Since the decision whether the traffic is to be blocked or allowed is taken by the user-agent, the kernel component must wait to receive the response from the user-agent to decide the fate of a connection. However, as kernel code, the kernel component cannot block and must take action on a packet before the user agent completes introspection. VMwall solves this problem for packets of unknown legitimacy by queuing the packets while waiting for the user agent's reply. When the user agent sends a reply, the module adds a rule for the connection. If the rule's action is to block the connection, then it drops all the packets that are queued. Otherwise, it re-injects all the packets into the network.

VMwall supplements the existing coarse-grained firewall provided by ebtables. Ebtables does not provide the ability to queue packets. Were it present, queuing would enable filters present inside the kernel to store packets for future processing

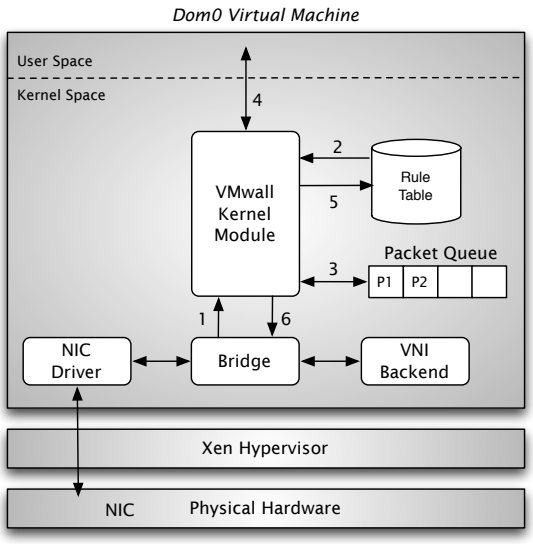


Figure 8: VMwall's kernel module architecture. (1) Packets inbound to and outbound from a guest domain are intercepted and passed to the kernel module. (2) The module receives each packet and looks into its rule table to find the rule for the packet. (3) The kernel module queues the packet if there is no rule present. (4) It sends an introspection request to the user agent and, after the agent completes, receives the dynamically generated rule for the packet. (5) The kernel module adds the rule into its rule table to process future packets from the same connection. (6) The kernel module decides based on the action of the rule either to accept the packet by reinjecting it into the network or to drop it from the queue.

and reinjection back into the network. To allow the VMwall kernel module to queue packets currently under inspection by the user agent, we altered ebtables to incorporate packet queuing and packet reinjection features. Figure 8 shows the design of the kernel component with the packet queuing and traffic filtering capability.

3.7.3 Modifications to the User Component

The user agent uses information about the process to create a firewall rule enforceable by the kernel component. The user agent maintains a whitelist of processes that are allowed to make network connections. When the user agent extracts the name of a process corresponding to the network packet, it searches the whitelist for the same name. VMwall allows the connection if it finds a match and blocks the connection otherwise. It then generates a rule for this connection and passes the rule to the

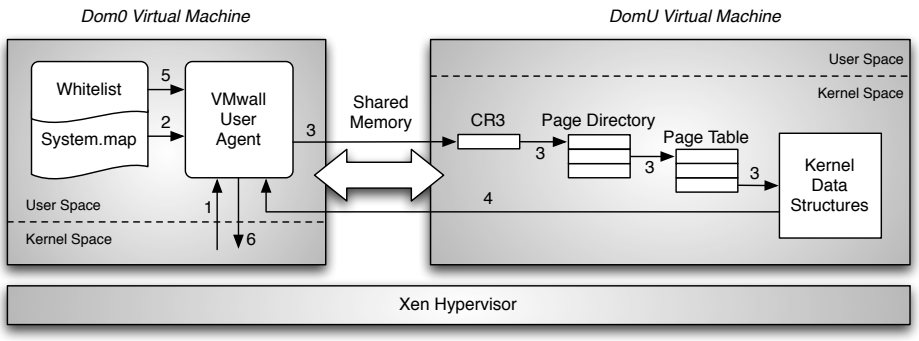


Figure 9: The user agent's architecture. (1) The user agent receives the introspection request. (2) The user agent reads the symbol file to extract the kernel virtual addresses corresponding to known kernel symbols. (3) The user agent uses Xen to map the domU kernel memory pages containing process and network data structures. (4) It traverses the data structures to correlate network and process activity. (5) The agent searches for the recovered process name in the whitelist. (6) The user agent sends a filtering rule for the connection to the kernel module.

VMwall kernel component. This rule contains the network connection information and either an allow or block action. The kernel component then uses this rule to filter subsequent packets in this attempted connection. Figure 9 shows the user agent design with the whitelist.

3.7.4 Evaluation

Though we performed detailed security and performance evaluation of our process attribution software, in this section we evaluated the ability of VMwall to filter out packets made by several different classes of attacks while allowing packets from known processes to pass unimpeded. We tested VMwall against Linux-based backdoors, worms, and bots that attempt to use the network for malicious activity. Table 4 reports the results of VMwall against attacks that receive inbound connections from attackers or connect out to remote systems. It can be seen that VMwall successfully blocked all inbound and outbound connections attempted by malware as these malicious processes were not in the whitelist. Table 5 reports results of tests using legitimate software in the presence of VMwall. It is evident from the results that

Table 4: Results of executing illegitimate software in the presence of VMwall. "Blocked" indicates that the network connections to or from the processes were blocked.

Name	*Connection Type*	*Result*
Blackhole	Inbound	Blocked
Gummo	Inbound	Blocked
Bdoor	Inbound	Blocked
Ovas0n	Inbound	Blocked
Cheetah	Inbound	Blocked
Apache-ssl	Outbound	Blocked
Apache-linux	Outbound	Blocked
Kaiten	Outbound	Blocked
Q8	Outbound	Blocked
BackDoor-Rev.b	Outbound	Blocked
Coromputer Dunno	Outbound	Blocked

Table 5: Results of executing legitimate software in the presence of VMwall. "Allowed" indicates that the network connections to or from the processes were passed as though a firewall was not present.

Name	*Connection Type*	*Result*
rcp	Outbound	Allowed
rsh	Outbound	Allowed
yum	Outbound	Allowed
rlogin	Outbound	Allowed
ssh	Outbound	Allowed
scp	Outbound	Allowed
wget	Outbound	Allowed
tcp client	Outbound	Allowed
putty	Outbound	Allowed
Internet Explorer	Outbound	Allowed
thttpd	Inbound	Allowed
tcp server	Inbound	Allowed
sshd	Inbound	Allowed

VMwall allowed legitimate software to function unimpedingly.

The design of VMwall differs from the process attribution software as VMwall's kernel component queues packets until it gets a response for the user-agent. This design incurs overhead on network applications running inside user VMs. To measure VMwall's performance overhead, we performed experiments with two different metrics

Table 6: Time (seconds) to transfer a 175 MB file between dom0 and domU, with and without VMwall.

Direction	*Without VMwall*	*With VMwall*	*Overhead*
File Transfer from Dom0 to DomU	1.105	1.179	7%
File Transfer from DomU to Dom0	1.133	1.140	1%

Table 7: Single TCP connection setup time (μs) measured both with and without VMwall inside dom0.

Direction	*Without VMwall*	*With VMwall*	*Overhead*
Connection from Dom0 to DomU	197	465	268
Connection from DomU to Dom0	143	1266	1123

for both inbound and outbound connections. In the first experiment, we measured VMwall's impact on network I/O by transferring a 175 MB video file over the virtual network via `wget`. Our second experiment measured the time necessary to establish a TCP connection or transfer UDP data round-trip as perceived by software in domU.

We first transferred the video file from dom0 to domU and back again with VMwall running inside dom0. Table 6 shows the result of our experiments. The median overhead imposed by VMwall is less than 7% when transferring from dom0 to domU, and less than 1% when executing the reverse transfer.

Our second metric evaluated the impact of VMwall upon connection or data stream setup time as perceived by applications executing in domU. For processes using TCP, we measured both the inbound and outbound TCP connection setup time. For software using UDP, we measured the time to transfer a small block of data to a process in the other domain and to have the block echoed back.

We created a simple TCP client-server program to measure TCP connection times. The client program measured the time required to connect to the server, shown in Table 7. Inbound connections completed quickly, exhibiting median overhead of only 268 μs. Outbound connections setup from domU to dom0 had a greater median overhead of 1123 μs, due directly to the fact that the introspection time for outbound

Table 8: Single UDP echo-reply stream setup time (μs) with and without VMwall. In an inbound-initiated echo, dom0 sent data to domU and domU echoed the data back to dom0. An outbound-initiated echo is the reverse.

Direction	*Without VMwall*	*With VMwall*	*Overhead*
Inbound Initiated	434	815	381
Outbound Initiated	271	848	577

connections is also high. Though VMwall's connection setup overhead may look high as a percentage, the actual overhead remains slight. Moreover, the introspection cost occurring at connection setup is a one-time cost that gets amortized across the duration of the connection.

We lastly measured the time required to transmit a small block of data and receive an echo reply to evaluate UDP stream setup cost. We wrote a simple UDP echo client and server and measured the round-trip time required for the echo reply. Note that only the first UDP packet required introspection; the echo reply was rapidly handled by a rule in the VMwall kernel module created when processing the first packet. We again have both inbound and outbound measurements, shown in Table 8. The cost of VMwall is small, incurring slowdowns of 381 μs and 577 μs, respectively.

VMwall currently partially optimizes its performance, and additional improvements are clearly possible. VMwall performs introspection once per connection so that further packets from the same connection are allowed or blocked based on the in-kernel rule table. VMwall's performance could be improved in future work by introducing a caching mechanism to the introspection operation. The VMwall introspection routine traverses the guest OS data structures to perform correlation. In order to traverse a data structure, the memory page that contains the data structure needs to be mapped, which is a costly operation. One possible improvement would be to support caching mechanisms inside VMwall's user agent to cache frequently used memory pages to avoid costly memory mapping operations each time.

3.8 Conclusion

We set out to design a process attribution software resistant to the direct attacks that bring down these security utilities today. Our software remains protected from attack by leveraging virtual machine isolation. Although it is a distinct virtual machine, it can recover process-level information of the vulnerable system by using virtual machine introspection to correlate network flows with processes bound to those flows. We have shown the efficacy of our software by correctly attributing network-level connections to host-level processes bound to those connections. Our connection detection operates with reasonable overheads upon system performance. We also showed another application of process attribution by designing a novel tamper-resistant application aware firewall that is effective in blocking backdoor, bot, and worm traffic emanating from the monitored system.

CHAPTER IV

IDENTIFICATION OF PROCESS MANIPULATION

4.1 Motivation

The process attribution software presented in Chapter 3 determines the end-point of a malicious connection inside the infected system. Though knowing this information identifies the presence of malware, it is still not sufficient to identify all malicious code present on infected systems. The process attribution alone cannot determine if an identified process is itself malicious, or if it is a hijacked benign victim of parasitic attacks. Parasitic behaviors help malware instances perform activities—such as spam generation, denial-of-service attacks, and propagation—without themselves raising suspicion.

In parasitic attacks, malicious software subverts the normal execution of benign processes by modifying their in-memory code image. For example, the conficker worm injects undesirable dynamically linked libraries (DLLs) into legitimate software [149]. In another example, the storm worm injects code into a user-space network process from a malicious kernel driver to initiate a DDoS attack from the infected computers [104]. To identify and eradicate all malicious code present on infected systems, it is important both to terminate maliciously-acting but benign processes and to find parasitic malware that may have induced the malicious activity.

A process can suffer from parasitic behaviors from either another process or an untrusted kernel driver. This chapter presents the detailed description of parasitic behaviors occurring at user- and kernel-level. We describe different methods of launching parasitic attacks using interfaces designed for benign purposes. This chapter then presents the techniques and design of host attribution software for identifying

process-to-process parasitic behaviors in userspace. We model a process-to-process parasitic attack as sequences of Windows API calls and detect the attacks by monitoring non-bypassable system-calls invoked by processes. As shown in Figure 1, our host attribution software executes from the hypervisor, and hence remains protected from direct attacks.

To detect untrusted drivers' parasitic behaviors, we need an ability to monitor drivers' execution behaviors. Due to the complexity involved in designing this monitoring software, we present the details of our defenses and monitoring architecture for kernel-to-process parasitic attacks in Chapter 5.

4.2 Previous Approaches

Host-based security software generally either scans unknown programs for patterns that match signatures of known malware [21,74] or continually monitors behaviors of software searching for unusual or suspicious runtime activity [50, 61, 128]. Our host attribution software is closest in spirit to the latter systems. It monitors the execution behavior of processes and untrusted drivers to identify instances of DLL injection or remote thread injection. Unlike traditional host-based utilities, it does not rely on injection alone as evidence of malware, as benign software sometimes uses injection for benign purposes. A heuristic-based malware detection system that monitors system calls or kernel APIs and detects code injection attacks may produce false positives. For example, DLL injection is used by the Microsoft Visual Studio debugger to monitor processes under development. Likewise, the Google toolbar injects code into `explorer.exe` (the Windows graphical file browser) to provide Internet search from the desktop. The host attribution software uses system-call information only when a network-level intrusion detection provides corroborating evidence of an attack.

Backtracker [75] reconstructs the sequence of steps that occurred in an intrusion by using intrusion alerts to initiate construction of event dependency graphs. In a similar

way, the host attribution software uses NIDS alerts to initiate discovery of malicious software even in the presence of parasitic behaviors. Technical aspects of Backtracker and the host attribution software differ significantly. Backtracker identifies an attack's point of entry into a system by building dependencies among host-level events. It assumes that operating system kernels are trusted and hence monitors system calls; it stores each individual system call in its log for later dependency construction. The host attribution software monitors and stores only high-level parasitic behaviors. It does not trust the OS kernel and assumes that kernel-level malware may be present, and it monitors both system calls and kernel APIs to detect both user- and kernel-level parasitism. Both Backtracker and the host attribution software are useful to remediation in different ways: the host attribution software's information guides direct removal of malicious processes, while Backtracker's information helps develop patches or filters that may prevent future reinfection at the identified entry point.

Malware analysis tools [160] have also built upon virtualization. Dinaburg et al. [29] developed an analysis system that, among other functionality, traced the execution of system calls in a manner similar to our host attribution sensor. Martignoni et al. [91] proposed a system that built a model of high-level malware behavior based upon observations of low-level system calls. Like that system, the host attribution software uses a high-level characterization of DLL and thread injection identified via low-level system-call monitoring; however, our system does not employ the performance-costly taint analysis used by Martignoni. In contrast to analysis systems, our goal is to provide malware detection via correct attribution of malicious behavior to parasitic malware. We expect that it could act as a front-end automatically supplying new malware samples to deep analyzers.

Table 9: Different parasitic behavior occurring from user- or kernel-level.

Number	*Source*	*Target*	*Description*
Case 1A	Process	Process	DLL and thread injection
Case 1B	Process	Process	Raw code and thread injection
Case 2A	Kernel driver	Process	DLL and thread alteration
Case 2B	Kernel driver	Process	Raw code and thread alteration
Case 2C	Kernel driver	Process	Kernel thread injection

4.3 Parasitic Malware

The host attribution software discovers parasitic malware. In this section, we present the threat model under which the host attribution software operates and describe common parasitic behaviors exhibited by malware.

4.3.1 Threat Model

We developed the host attribution software to operate within a realistic threat model. We assume that attackers are able to install malicious software on a victim computer system at both the user and kernel levels. Installed malware may modify the system to remain stealthy. These facts are demonstrated by recent attacks happening at the user and the kernel level. A preventive approach that does not allow users to load untrusted drivers may not be effective because users sometimes unknowingly install untrusted drivers for various reasons, such as gaming or adding new devices.

4.3.2 Malware Behaviors

Parasitic malware alters the execution behavior of existing benign processes as a way to evade detection. These malware often abuse both Windows user and kernel API functions to induce parasitic behaviors. We consider a malware parasitic if it injects either executable code or threads into other running processes. The parasitic behaviors can originate either from a malicious user-level process or a malicious kernel driver. Table 9 lists the different cases in which malware can induce parasitic behavior,

and the following section explains each of those cases in detail.

Case 1A: *Dynamically-linked library (DLL) injection* allows one process to inject entire DLLs into the address space of a second process [124]. An attacker can author malicious functionality as a DLL and produce malware that injects the DLL into a victim process opened via the Win32 API call `OpenProcess` or created via `CreateProcess`. These functions return a process handle that allows for subsequent manipulation of the process. The malware next allocates memory inside the victim using the `VirtualAllocEx` API function and writes the name of the malicious DLL into the allocated region using `WriteProcessMemory`. Malware cannot modify an existing thread of execution in the victim process, but it can create a new thread using `CreateRemoteThread`. The malware passes to that function the address of the `LoadLibrary` API function along with the previously written-out name of the malicious DLL.

Case 1B: A *raw code injection* attack is similar to a DLL injection in that user-space malware creates a remote thread of execution, but it does not require a malicious DLL to be stored on the victim's computer system. The malware allocates memory space as before within the virtual memory region of the victim process and writes binary code to that space. It then calls `CreateRemoteThread`, passing the starting address of the injected code as an argument.

Case 2A: A kernel-level malicious driver also shows parasitic behavior by injecting malicious DLLs inside the user-space process. A malicious driver can perform this task in a variety of ways, such as by calling system call functions directly from the driver. A stealthy technique involves Asynchronous Procedure Calls (APCs): a method of executing code asynchronously in the context of a particular thread and, therefore, within the address space of a particular process [98]. Malicious drivers identify a process, allocate memory inside it, copy the malicious DLL to that memory, create and initialize a new APC, alter an existing thread of the target process to

execute the inserted code, and queue the APC to later run the thread asynchronously. This method is stealthy as APCs are very common inside the Windows kernel, and it is very hard to distinguish between benign and malicious APCs.

Case 2B: This method is similar to the one explained in Case 2A. The difference lies in the form of malicious code that injected into a benign process. Here, malicious kernel drivers inject raw code into a benign process and execute it using the APC.

Case 2C: Finally, a *kernel thread injection* is the method by which malicious drivers execute malicious functionality entirely inside the kernel. A kernel thread executing malicious functionality is owned by a user-level process, though the user-level process had not requested its creation. By default, these threads are owned by the `System` process, however a driver may also choose to execute its kernel thread on behalf of any running process.

Our system adapts well as new attack information becomes available. Though the described methods are prevalent in current attacks, other means of injecting malicious code into benign software also exist. For example, `SetWindowsHookEx`, `AppInit_DLL`, and `SetThreadContext` APIs can be used for malice. Our general technique can easily encompass these additional attack vectors by monitoring their use in the system. We describe the monitoring of user-level parasitic behaviors (Cases 1A & 1B) in the rest of the chapter and defer the monitoring of kernel-level parasitic behaviors (Cases 2A, 2B, & 2C) to Chapter 5.

4.4 User-level Parasitism

User-level parasitism occurs when a malicious user process injects a DLL or raw code along with a thread into a running benign process as explained in Cases 1A and 1B. To detect a process-to-process parasitic behavior, the host attribution software continuously monitors the runtime behavior of all processes executing within the victim VM by intercepting their system calls [39, 49]. Note that we monitor the

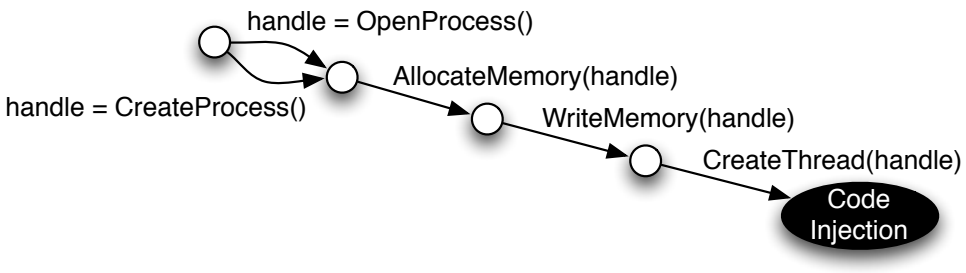


Figure 10: Runtime parasitic behavioral model for a process-to-process injection.

native API, or the transfers from userspace to kernel space, rather than the Win32 API calls described in Section 4.3.2. High-level API monitors are insecure and may be bypassed by knowledgeable attackers, but native API monitoring offers complete mediation for user-level processes.

The host attribution software intercepts all system calls, but it processes only those that may be used by a DLL or thread injection attack. This list includes `NtOpenProcess`, `NtCreateProcess`, `NtAllocateVirtualMemory`, `NtWriteVirtual-Memory`, `NtCreateThread`, and `NtClose`, which are the native API forms of the higher-level Win32 API functions described previously. Our software records the system calls' parameter values: *IN* parameters at the entry of the call and *OUT* parameters when they return to userspace. Recovering parameters requires a complex implementation that we describe in detail in Section 4.5.

The host attribution software uses an automaton description of malware parasitism to determine when DLL or thread injection occurs. The automaton (Figure 10) characterizes the series of system calls that occur during an injection. As the host attribution software intercepts system calls, it verifies them against an instance of the automaton specific to each possible victim process. We determine when the calls apply to the same victim by performing data-flow analysis on the process handle returned by `NtOpenProcess` and `NtCreateProcess`. Should the handle be duplicated (using `NtDuplicateObject`), we include the new handle in further analysis.

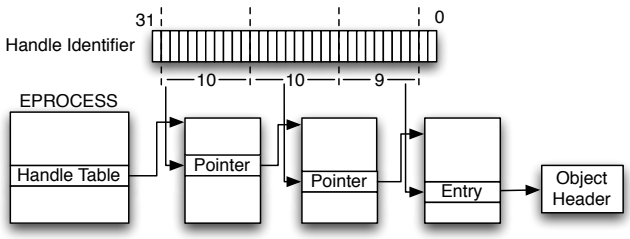


Figure 11: Handle resolution in Windows converts a 32-bit handle identifier into a structure for the object referenced by the handle. Resolution operates in a manner similar to physical address resolution via page tables.

4.5 Low-Level Implementation Details

The host attribution software is an operating prototype implemented for Windows XP SP2 victim systems hosted in virtual machines by the Xen hypervisor version 3.2. The high-privilege VM executing our software runs Fedora Core 9. Implementing the host attribution software for Windows victim systems required technical solutions to challenging, low-level problems.

4.5.1 Resolution of Handle to Process

It is required to identify the names of processes that receive DLL or thread injection from other, potentially malicious, software. The host attribution software observes *IN* parameters to the Windows system calls used as part of an injection, and these parameters include handles. All handles used by a process are maintained by the Windows kernel in handle tables, which are structured as shown in Figure 11.

To resolve a handle to a process name, our software uses the handle to find the corresponding `EPROCESS` data structure in the Windows kernel memory. Since we know the process ID of an injecting process, the host attribution software can find that process' handle table. It searches the table for the specific object identifier recorded by the system. As a pleasant side-effect, this inspection of the handle table will additionally reveal the collection of files and registries currently open to the possibly malicious injecting process.

4.5.2 System Call Interpositioning and Parameter Extraction

The host attribution software requires information about system calls used as part of DLL or thread injection. We developed a system call interpositioning framework deployable in Xen; this framework supports inspection of both *IN* and *OUT* system call parameters. An *IN* parameter's value is passed by the caller of a system call while an *OUT* parameter's value is filled after the execution of the system call inside the kernel.

Windows XP uses the fast x86 system-call instruction `SYSENTER`. This instruction transfers control to a system-call dispatch routine at an address specified in the `IA32_SYSENTER_EIP` register. Unfortunately, the Intel VTx hardware virtualization design does not allow the execution of `SYSENTER` to cause a VM to exit out to the hypervisor. As a result, our host attribution software must forcibly gain execution control at the beginning of a system call. It alters the contents of `IA32_SYSENTER_EIP` to contain a memory address that is not allocated to the guest OS. When a guest application executes `SYSENTER`, execution will fault to the hypervisor, and hence to our code, due to the invalid control-flow target.

Inside the hypervisor, our software processes all faults due to its manipulation of the register value. It records the system call number (stored in the `eax` register), and it uses the `edx` register value to locate system-call parameters stored on the kernel stack. The host attribution software extracts *IN* parameters with a series of guest memory read operations. It uses the `FS` segment selector to find the *Win32 thread information block* (TIB) containing the currently-executing process' ID and thread ID. It then modifies the instruction pointer value to point at the original address of the system-call dispatch routine and re-executes the faulted instruction.

We use a two-step procedure to extract values of *OUT* parameters at system-call return. In the first step, we record the value present in an *OUT* parameter at the beginning of the system call. Since *OUT* parameters are passed by reference,

the stored value is a pointer. In order to know when a system call's execution has completed inside the kernel, we modify the return address of an executing thread inside the kernel with a new address that is not assigned to the guest OS. This modification occurs when intercepting the entry of the system call. In the second step, a thread returning to usermode at the completion of a system call will fault due to our manipulation. As before, the hypervisor receives the fault. The host attribution software reads the values of *OUT* parameters, restores the original return address, and re-executes the faulting instruction. By the end of the second step, the host attribution software has values for both the *IN* and *OUT* system-call parameters.

4.6 Evaluation

We tested our prototype implementation of the host attribution software to evaluate its ability to appropriately identify parasitic attacks on infected systems and its performance.

4.6.1 User-level Malware Identification

We tested the host attribution software's ability to detect process-to-process parasitic behaviors with the recent conficker worm [149]. Conficker employs DLL injection to infect benign processes running on the victim system. We executed conficker inside a test VM monitored by the host attribution software. When executed, the worm ran as a process called `rundll32.exe`. During the execution of the created process, it issued various systems calls, including the ones that launched parasitic attacks on benign processes. The host attribution software monitored all system calls and recorded those that matched the model of a process-to-process injection. With the help of data flow analysis, the host attribution software recorded a DLL injection behavior from `rundll32.exe` targeting benign specific `svchost` processes. We repeated these tests with the `Adclicker.BA` trojan and successfully detected its parasitic behavior also.

Table 10: Results of CPU performance tests for unmonitored execution and for the host attribution software's monitoring with and without parasitic behaviors present; higher absolute measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Parasitic Behavior*			
		Present	%	*Absent*	%
Integer Math (MOps/sec)	126.5	92.5	26.88	124.8	1.34
Floating Point Math (MOps/sec)	468.4	439.5	6.17	444.3	5.14
Compression (KB/sec)	1500.9	1494.7	0.41	1496.0	0.32
Encryption (MB/sec)	4.21	4.19	0.48	4.20	0.24
String Sorting (Thousand strings/sec)	1103.3	1072.2	2.82	1072.3	2.81

Table 11: Results of memory performance tests for unmonitored execution and for the host attribution software's monitoring with and without parasitic behaviors present; higher absolute measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Parasitic Behavior*			
		Present	%	*Absent*	%
Allocate Small Block (MB/sec)	2707.4	2322.3	14.22	2704.1	0.12
Write (MB/sec)	1967.0	1931	1.83	1942.9	1.23

4.6.2 Performance

We designed the host attribution software to operate at runtime, so its performance cost on an end user's system must remain low. We tested our prototype on an Intel Core 2 Quad 2.66 GHz system. We assigned 1 GB of memory to the untrusted Windows XP SP2 VM and 3 GB combined to the Xen hypervisor and the high-privilege Fedora Core 9 VM. We carried out CPU and memory experiments using a Windows benchmark tool called `PassMark Performance Test` [110]. Our experiments measured the host attribution software's overhead during benign operations and during active parasitic attacks. We executed all measurements five times and present here the median values.

We measured the host attribution software's overhead on CPU-bound and memory intensive operations. Tables 10 and 11 list a collection of benchmark measurements for execution in a VM with and without the host attribution software's monitoring. For executions including the host attribution software, we measured performance both

during execution of a DLL injection attack against an unrelated process and during benign system operation. As shown in the tables, the performance of our software in the absence of parasitic behavior is excellent and largely reflects the cost of system-call tracing. Experiments including the execution of an injection attack show diminished performance that ranges from inconsequential to a more substantial performance loss of 27%. The additional overhead measured during the attack occurred when the host attribution software identified injection behavior and harvested state information for its log. This overhead is infrequent and occurs only when parasitic behaviors actually occur. These results show that our host attribution software is performant and capable of detecting real world parasitic attacks without impacting the system.

4.7 Conclusion

We presented techniques and a prototype of the host attribution software that detects parasitic attacks. Our host attribution software discovered malicious code that launches parasitic attacks on benign processes by monitoring the system behaviors from the hypervisor. Real malware samples showed that the host attribution software correctly identified malicious process-to-process parasitic attacks. Our performance analysis demonstrated that our solution was suitable for real world deployment.

CHAPTER V

KERNEL MODE MONITORING

5.1 Motivation

The previous chapter described our defenses for a process-to-process parasitic attack. As mentioned in Section 4.3.2, parasitic attacks are also caused by malicious kernel drivers. Kernel-level parasitism occurs when a malicious kernel driver injects either a DLL or raw code followed by the alteration of an existing targeted process' thread (as described in Cases 2A and 2B of Chapter 4), or by creating a new thread owned by any process as explained in Case 2C. To detect kernel-to-process parasitic behaviors, we need to monitor all kernel APIs invoked by malicious drivers.

Monitoring execution of kernel drivers is challenging due to the lack of any driver monitoring interface inside the kernel. The lack of a memory barrier inside operating systems allows kernel-mode malware to directly invoke arbitrary kernel functionality by simply calling or jumping to arbitrary kernel code addresses. Though commodity operating systems publish interfaces to be used by drivers and loadable modules to request services from the kernel, there is no mechanism to enforce the implicitly trusted boundary between the core kernel and its drivers.

The key contribution of this chapter is to augment the host attribution software with a kernel API monitor that monitors all kernel APIs invoked by drivers and ensures complete mediation of direct accesses from drivers to kernel code. Our kernel API monitor creates distinct virtual memory regions for commodity monolithic kernels and their drivers in the same way that kernels manage distinct regions for higher-level application software. This design isolates drivers in a different memory region and requires them to invoke kernel code through published entry points,

allowing the monitor to record all invoked kernel APIs.

5.2 Previous Approaches

Our software hardens the kernel API to drivers by isolating driver code in a different memory address space than the kernel. Previous researchers have studied the *extension isolation problem* from both the fault isolation and security perspectives, and they proposed solutions different than ours. Nooks [145] confined drivers to a separate address space using hardware page protection. The goal of Nooks was fault isolation; a malicious driver could easily bypass its protection. Since Nooks resided in the operating system, it was still susceptible to direct attacks by malicious kernel drivers. Further, it required assistance from both drivers and the kernel, and its overhead was high ($\approx 60\%$). In contrast, we designed the kernel API monitor to offer its protection from all drivers, including malware, and it protects itself by using the hypervisor. Though the kernel API monitor also isolates drivers using hardware protection, it has low overhead due to its fast address space switching. Vx32 [38] and NaCl [168] isolated applications in sandboxed environments to execute them safely. They used segmentation and programming language techniques to prevent applications from breaking out of the sandbox. The kernel API monitor is different from Vx32 and NaCl as it isolates kernel extensions, protecting the core kernel. Further, it uses paging and binary rewriting to reduce the performance overhead.

Other solutions isolate drivers in user space. Ganapathy et al. [44] proposed the Microdrivers architecture in which drivers were broken into two components, one residing in kernel-space and the other in user-space. Though Microdrivers improved the reliability of the system, the drivers were not completely isolated from the kernel. Nexus [162] isolated drivers entirely in user-space using hardware protection mechanisms. However, it required additional device-specific safety specifications. Though these approaches have merits, they require extensive code changes and rewriting of

all drivers.

Researchers have also explored protection domains implemented entirely in software. Software fault isolation (SFI) [157] used program rewriting techniques to modify the object code of untrusted modules to prevent them from writing or jumping to an address outside their access domain. The program rewriting technique was further used by XFI [32]. XFI guarded all instructions to prevent control flow and data access violations. The control flow prevention included entry point protection similar to the kernel API monitor's protection. XFI's rewriter assumed either the availability of debugging information (PDB files) associated with drivers or specially-compiled drivers. Neither assumption is valid for malware: malware instances deliberately strip debugging information to make their analysis hard and use packing software to further hide the difference between code and data. It is not feasible to force malware authors to use special compilers. In contrast, the kernel API monitor does not assume any cooperation from drivers, hence it is suitable for protection from malware.

Our monitor uses memory page protection bits in its creation of memory barriers. Payne et al. [113] used page protection bits to protect trampoline code inserted into kernel memory. SecVisor [130] protected the core kernel code pages from modification by kernel-level malware. Litty et al. [86] identified covertly executing rootkit binaries present on an infected system by using page protection bits to intercept any code execution attempt involving protected pages. Sharif et al. [131] presented an in-VM design of a security monitor by isolating the security driver in a separate memory region using hardware page protection. Chen et al. [18] proposed a system based on multi-shadowing that protected the privacy and integrity of an application, even if the underlying operating system was compromised. Similar to these systems, the kernel API monitor also uses page protection bits, here, to create a non-bypassable interface inside the kernel for drivers.

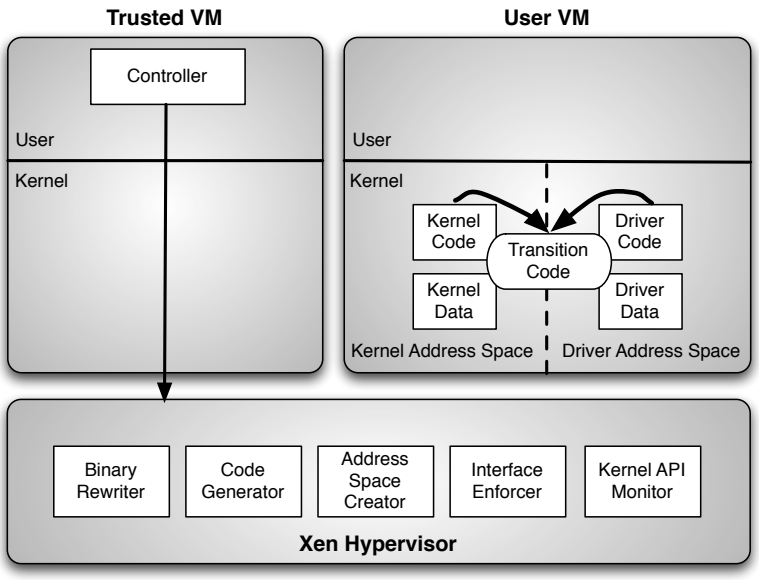


Figure 12: High-level architecture of the kernel API monitor.

5.3 Non-Bypassable Kernel Interface to Drivers

We designed and developed the kernel API monitor, shown in Figure 12, to fulfill the following goals:

- **Kernel API Monitoring:** As malware instances contain all malicious functionalities in drivers, we need a security monitor that can monitor kernel interfaces provided to drivers.

- **Kernel Interface Enforcement:** The kernel API monitor requires a non-circumventable kernel interface to monitor drivers' interactions with the kernel. Our software creates a non-bypassable interface by isolating drivers in a separate address space and allowing kernel invocations only via predefined entry points.

- **Efficiency:** The techniques used to enforce a non-bypassable interface to the kernel must not introduce serious performance impediments to normal system usage. The kernel API monitor performs on-demand dynamic binary rewriting and code generation, a design that keeps overhead low.

5.3.1 Driver Isolation

Commodity operating systems such as Windows and Linux rely on paging to provide address space isolation between user applications and the kernel. Page tables, a data structure defined by the hardware, map virtual addresses to physical addresses. Each process has its own page tables (virtual address space), each of which contains mappings for all kernel virtual memory addresses at a fixed location. The kernel page mappings include all kernel-mode components. Our approach creates separate page tables for the kernel and for drivers, much in the same way that an OS kernel creates distinct page tables for each running process. Separate page tables force all control flows spanning the kernel-driver interface to induce page faults handled by code in the hypervisor that verifies the legitimacy of the control flow.

In virtualized environments, the hypervisor controls the machine's memory by creating its own page tables to be used by the memory management hardware. These hypervisor-level tables are called shadow page tables in virtualization literature and active page tables (APTs) by hardware vendors [65]. We refer to the portion of the shadow page tables that translates kernel virtual addresses to kernel physical addresses as the kernel page table (KPT). The page tables present in the guest VM—normally used in the absence of virtualization—are renamed as virtual page tables and provide the illusion to the guest kernel that it controls its own memory.

The kernel API monitor creates a separate *driver* address space inside the Xen hypervisor analogous to the existing kernel address space. It maps all driver code pages from all drivers loaded by the guest system into the driver page table (DPT) in a manner transparent to the guest. To maintain consistency between the DPT and the KPT, we map all memory pages of the kernel address range into both page tables, though we set permissions differently. In the KPT, driver code pages are marked non-executable and non-writable. In the DPT, kernel code pages are marked

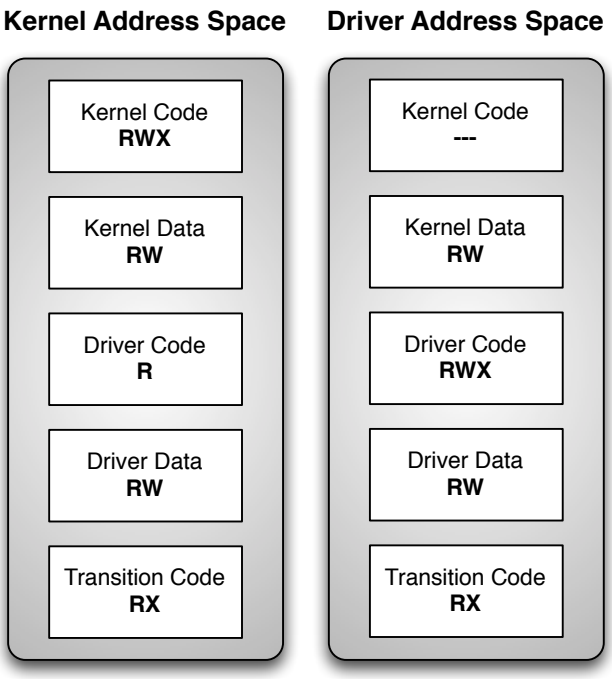


Figure 13: Layout of kernel and driver address spaces with permissions set on memory pages.

non-executable, non-readable, and non-writable. We also mark all data pages non-executable in both the KPT and DPT to prevent drivers from using data pages for code execution (Figure 13). This configuration of execution permissions across the page tables sets up our subsequent monitoring of a driver's invocation of kernel functionality. Both tables contain executable and non-writable transition code pages, described subsequently in Section 5.4. Both the KPT and DPT are stored in the hypervisor's memory space, and hence remain isolated from attack by malicious software executing in a guest VM.

To map driver code pages into the DPT, the kernel API monitor must know the virtual addresses of driver code pages in the guest kernel's memory space. Since the operating system loads drivers and modules dynamically, they are relocatable and do not reside at a fixed location at every load. To acquire the addresses at which drivers are loaded, we interpose on the driver loading process. At runtime, we automatically

rewrite the target of a direct call instruction along the driver loading path inside the kernel to point to a memory location that is not mapped to the guest VM; we store the correct target location inside the hypervisor. This design creates a page fault during driver loading, allowing the kernel API monitor to gain control. Since this fault happens after the OS decides where to load a driver, the kernel API monitor extracts this address information from the guest memory and then resumes the guest's execution.

We include the (non-executable) driver code pages in the KPT for simple efficiency purposes. Some drivers contain read-only data, such as the import table for the kernel, and executable code on the same memory page. By including the drivers' code pages in the KPT, the kernel can still read the drivers' read-only data without introducing extra page faults. In contrast, kernel code pages must be unreadable in the DPT to prevent introduction of a code-pullout attack [3, 73] and to inhibit return-oriented rootkits. A return-oriented rootkit [63] requires gadgets, many taken from the core kernel, to perform arbitrary computation. The kernel API monitor limits return-oriented programming by forcing drivers to enter the kernel at legitimate API entry points; attackers are unable to construct kernel gadgets different than the kernel functions themselves. Though attackers could construct gadgets from the code of other drivers in the DPT, we discuss in Section 5.5 an extension to the kernel API monitor that hardens and monitors the interface between drivers to remove even that opportunity.

5.3.2 Address Space Switching

The kernel API monitor changes address spaces by switching between the kernel and driver page tables. The root of the page tables is called the *page directory*, and in x86 architectures, the hardware register *CR3* stores the physical address of the current active page directory. In a virtualized environment, the hardware CR3 register points

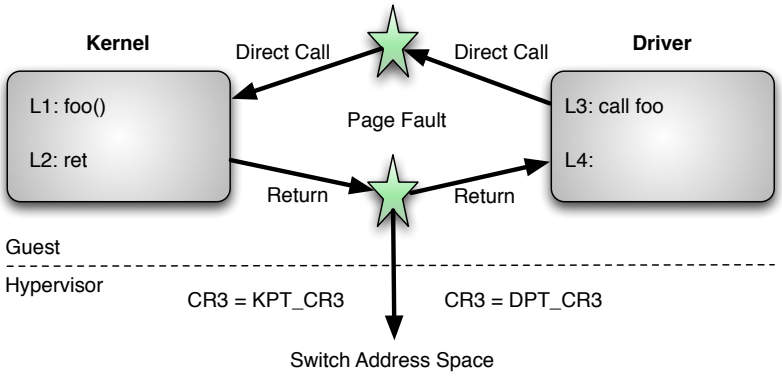


Figure 14: Address space switching between drivers and the kernel on the invocation of a direct call from the driver (slow path).

to the root of the shadow page tables managed by the hypervisor. Further, read and write instructions involving the CR3 register are also privileged operations. A guest operating system is not allowed to write into the CR3 register, so any write operation by guests to the CR3 register causes a world switch (transition to the hypervisor) that passes control to the hypervisor. This feature thwarts attacks in which malware running in the kernel attempts to modify the CR3 register to point to new page tables.

The kernel API monitor switches between two address spaces by changing the value stored in the CR3 register. During driver code execution, the CR3 register stores the DPT's root address, `DPT_CR3`. When the driver code calls or jumps into any kernel code, the execution faults into the hypervisor due to the page protection bits set on the kernel code pages in the DPT. Inside the hypervisor, the kernel API monitor intercepts the fault, and if the fault is for the kernel code pages, it changes the CR3 value stored in the register by using the KPT's root address, `KPT_CR3`. After changing the CR3 value in the hypervisor, the kernel API monitor returns to the guest OS to re-execute the faulted instruction. On the return path from the core kernel to drivers, execution faults again due to the page permission bits set on the driver code pages in the KPT. In this case, the kernel API monitor replaces the CR3 register's value with the value of `DPT_CR3`. Figure 14 describes the address space

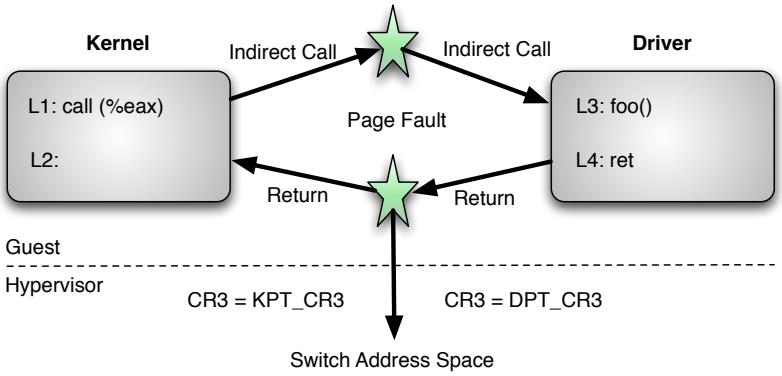


Figure 15: Address space switching between the kernel and drivers on the invocation of an indirect call from the kernel (slow path).

switching process on a direct `call` to and `ret` from the kernel. The kernel API monitor performs similar switching on an indirect `call` to and `ret` from drivers code (Figure 15).

5.3.3 Non-Bypassable Interface Enforcement

The creation of a separate KPT and DPT with distinct page access permissions allows the kernel API monitor to intercept control flows from any driver to the kernel, and hence to limit allowed control flows to only those targeting valid API entry points in the kernel. Commodity operating systems publish interfaces meant to be used by third party developers creating drivers and loadable modules. Our expectation is that any interaction with the core kernel through these interfaces is legitimate and should be allowed, but attempts by a driver to jump or call other kernel addresses represents illicit behavior attempting to bypass the kernel interface. The kernel API monitor enforces the use of these interfaces upon drivers.

The kernel API monitor verifies calls and jumps from driver code to kernel code, returns from the driver to the kernel following the kernel's invocation of driver functionality, and interrupts. When driver code executing from the DPT invokes a kernel function, the system's execution will fault into the hypervisor because the kernel code in the DPT does not have execute permission. In the hypervisor, the kernel API

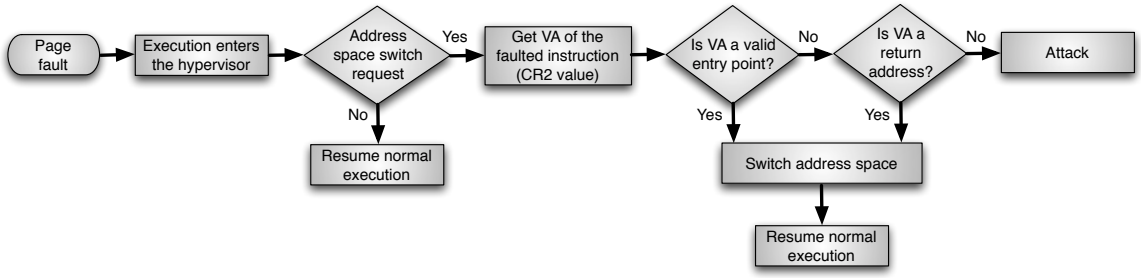


Figure 16: Steps involved in the verification of interface invocation from drivers to the core kernel.

monitor extracts the guest VM's execution context information, such as the virtual address of the faulted instruction. It verifies whether the faulted address corresponds to a predefined valid entry point into the kernel code—these include entry points of exported functions, interrupt handlers, and exception handlers. If the entry point is legitimate, the kernel API monitor alters the CR3 to specify the KPT as the current page table. If the faulted address is not a valid entry point into the kernel, then either the driver is attempting to invoke a kernel function which is not meant to be used by drivers, or it is trying to jump into the middle of a block of code. The kernel API monitor prevents such illicit control flow.

The kernel API monitor similarly verifies control flows that return back to the core kernel code upon execution of the `ret` instruction inside a driver. This driver-to-kernel transition is valid provided that the return address on the kernel's call stack has not been altered. At the original call from the kernel to the driver, the kernel API monitor records the return address at the top of the stack prior to switching the page tables from KPT to DPT. When the subsequent return instruction faults, the kernel API monitor then verifies whether the fault location matches the previously stored return address. This design defeats attacks in which attackers modify the return address to return to an arbitrary location in the kernel code. Figure 16 shows the steps involved during the verification of a control flow transition from drivers to the core kernel.

A malicious driver may attempt to use DMA to write memory mapped into the KPT and then to execute that code in the kernel's context. The kernel API monitor prevents malicious DMA writes by verifying the targeted memory regions in the requested DMA operations. Xen emulates all guest DMA operations using a software IOMMU. The kernel API monitor intercepts all DMA requests and rejects any request that contains an address not writable in the DPTs such as the kernel code and transition code pages. Xen also virtualizes recent hardware IOMMUs, such as Intel's VT-d and AMD's DEV. Protection from these DMA requests requires address verification at the virtualized hardware [26, 130].

The kernel API monitor must be aware of legitimate entry points into the kernel. We extract the virtual address of all kernel functions available to drivers for legitimate use from the symbol file (System.map or kallsyms in Linux and Debug Symbols in Windows) maintained by the guest kernel and keep this information with the kernel API monitor in the hypervisor.

5.3.4 Driver Page Table Implementation

We developed the driver address space as new shadow page tables created in the Xen hypervisor. Equally suitable alternative implementations could use other hypervisors, such as KVM or VMware, and hardware-supported nested/extended page tables. Although the guest Linux system used for our prototype development is a 32-bit system with 2-level page tables, we used Xen in its physical address extension (PAE) mode. In PAE mode, the x86 memory management unit expects 3-level page tables and offers the non-execute (NX) memory page permission absent from 2-level page tables. In PAE mode, Xen automatically maps 2-level guest page tables to its 3-level shadow page tables. The kernel API monitor then creates the DPT by allocating memory for a 3-level page table separate from Xen's original table. After allocating memory, it sets up the DPT as a copy of the shadow page table with kernel code

marked non-executable, non-writable, and non-readable. Finally, it edits the KPT so that driver code pages are marked non-executable and non-writable.

5.3.5 Persistent Protection

The kernel API monitor ensures that address spaces remain isolated throughout the guest system's execution. It thwarts attacks that attempt virtual memory remapping or creation of new memory mappings that reintroduce executable kernel code into the DPT and vice versa. The kernel API monitor ignores such requests and injects a page fault into the guest, indicating that the region is not for mapping. Our monitor utilizes the hypervisor's ability to interpose on the guest VM's virtual page table updates. It prevents the guest OS from mapping or changing protections on protected memory pages by hooking inside Xen's page table propagation `sh_propagate` and page fault handler `sh_page_fault` code. On each page fault, it verifies that the page protection bits have not been altered.

5.4 Fast Address Space Switching

The isolation of driver code pages in an address space separate from kernel code pages comes at a price. Each address space transition causes page faults, which in turn cause hypervisor world switches. Since the interaction between the kernel and drivers happens at a high rate, we expect the performance cost to be high. To this end, we propose a novel approach that reduces the transition overhead by establishing a fast path for address space switching. In this section, we describe the design and implementation of the fast path.

Our performance improvement comes by dramatically reducing the number of world switches that occur during guest system execution. In the design as presented in the previous section, every call and return spanning the barrier between the kernel and drivers induces a world switch to the hypervisor at the page fault. In our fast path design, only the first call at a particular call site faults to the hypervisor. All

subsequent calls from the same location and all corresponding returns execute at full speed. This design is similar to lazy linking of library functions in dynamically-linked applications: the first invocation executes functionality that fixes up the code so that all subsequent calls execute with no delay. Our fixups include runtime code generation and selective rewriting of guest kernel and driver code. A further optimization to prevent execution faults on even the first call instruction would require altered compilation or pre-execution offline code rewriting at all control-flow transfers, and we have not pursued such changes.

We leverage a hardware feature present in x86 processors called *CR3-Target Controls* [65]. This feature allows a guest kernel to change the CR3 value without causing a world switch to the hypervisor, provided that the value written into the CR3 register was previously specified by the hypervisor in the CR3-Target registers. The kernel API monitor adds the `KPT_CR3` and `DPT_CR3` values into the registers.

It is then the responsibility of guest kernel code to switch the CR3 value when transitioning the memory barrier between the kernel and the drivers. The instructions to execute the switch are not present in the stock guest kernel. The kernel API monitor thus generates short sequences of instructions that correctly change the CR3 value, writes those sequences into guest OS memory pages that we term *transition pages*, and overwrites call instructions in the kernel code and driver code to redirect the control flows spanning the memory barrier through the transition pages. The transition pages are guest memory pages, and they are mapped into both the KPT and DPT as read-only and executable pages. We call the short sequence of instructions on transition pages *transition code*.

In the hypervisor, the kernel API monitor generates transition code and rewrites call instructions on-demand at runtime every time a call instruction executes for the first time and faults to the hypervisor. Subsequent to the code alteration, execution of the same call instruction will pass through the transition code and avoid a world

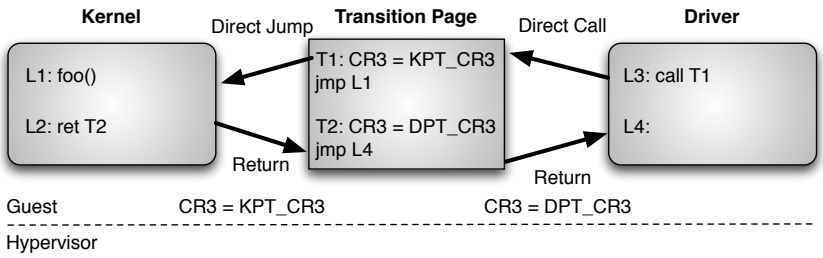


Figure 17: Address space switching between the kernel and drivers on the invocation of a direct call from a driver (fast path).

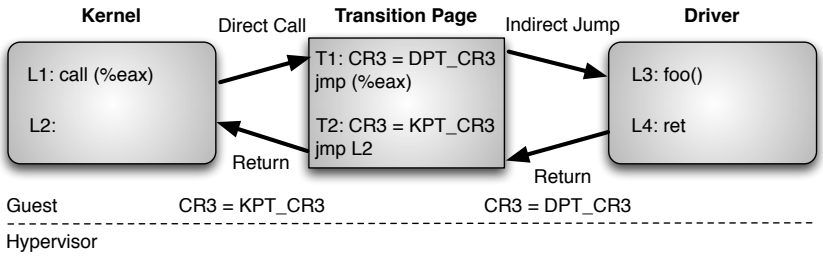


Figure 18: Address space switching between the kernel and drivers on the invocation of an indirect call from the kernel (fast path).

switch. We only redirect direct call instructions from drivers to the core kernel, indirect call instructions from the kernel to drivers, and their corresponding returns; indirect call instructions from drivers to the kernel and direct call instructions from the kernel to drivers still use the slow path for switching between address spaces (see Section 5.4.2 for explanation). By adding the `KPT_CR3` and `DPT_CR3` values to the CR3-Target registers, performing runtime code generation on transition pages, and rewriting control transfer instructions, any CR3 switch between the KPT and the DPT happens at native speed without invoking the hypervisor. Figures 17 and 18 show the effect of the fast path on direct and indirect call instructions occurring through the interface.

Our fast path design is secure because the kernel API monitor only overwrites those `call` instructions that bring legitimate control flows into the kernel at a valid API entry point. All transitions that have not been overwritten still fault, and the

kernel API monitor verifies those transitions. This verification is sufficient to guarantee the non-bypassable interface enforcement. Since transition pages are read-only, attackers cannot modify the generated code on transition pages to enter into arbitrary locations inside the kernel. The transition code is the only code that is executable in both the KPT and DPT, and it is the only way of switching the address spaces without invoking the hypervisor. Malicious drivers executing from the DPT cannot execute kernel code in the KPT by changing the CR3 to `KPT_CR3` themselves without using the transition code. Though the CR3 switch will not fault as `KPT_CR3` is in the CR3-Target registers, the execution will fault on the driver's next instruction as that instruction is not executable from the KPT.

5.4.1 Runtime Transition Code Generation

The kernel API monitor generates transition code on transition pages at runtime to switch the CR3 register value. The transition code can be divided into two parts: the entry code and the exit code. The entry code corresponds to a `call` instruction while the exit code corresponds to the paired `ret` instruction. The kernel API monitor generates the entry and exit code customized for each `call` and corresponding `ret`. We describe entry and exit code for both the direct and indirect call instructions that the kernel API monitor overwrites.

The entry code for a direct call instruction from a driver to the kernel has three sequential components: (a) code that overwrites the return address on the stack with the address of the start of the paired exit code, (b) CR3 switch code, and (c) a jump to the original target address in the kernel. In a single instruction, the transition code overwrites the return address to redirect the subsequent return from the kernel back to the driver through the exit code on the transition page. Note that the kernel API monitor records the original return address before overwriting its value; the original value will be used when generating the exit code. The kernel API monitor then

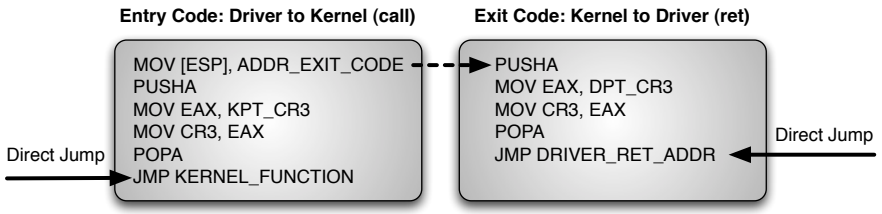


Figure 19: Runtime transition code generated by the kernel API monitor to enter in and exit from the kernel code on direct call and ret instructions, respectively.

generates the code that switches the address spaces without invoking the hypervisor, using a sequence of four instructions. When executed, this will not cause a page fault because the transition code is present in both the DPT and KPT. Finally, it adds a direct `jmp` instruction to jump to the original kernel function. Due to the address space switch that happens before the jump, this jump will not cause a page fault.

The paired exit code is similar to the entry code in that the generated code (a) switches the CR3 value back to `DPT_CR3` and then (b) jumps to the original return address in the driver. Since both the entry code and exit code are customized for each call and return, the direct `jmp` instructions use hardcoded values taken from the return address on the stack prior to its overwrite. The kernel API monitor writes these hardcoded values on the transition page at the time of the code generation. Figure 19 shows the transition code that the kernel API monitor generates for a direct call instruction from a driver to the kernel and for its return.

In a similar way, the kernel API monitor generates entry and exit code for indirect calls from the kernel to drivers. When producing entry code, the kernel API monitor first saves the original return address and generates code to replace it with the start of the exit code. Then, it generates code to switch the CR3 to `DPT_CR3`. In the next instruction, however, the jump target cannot be hardcoded because the indirect target may change later in execution. The address of the targeted driver function is either in a register or in a memory location specified as the operand of the call instruction. In order to ensure that the transition code targets the correct address, the kernel

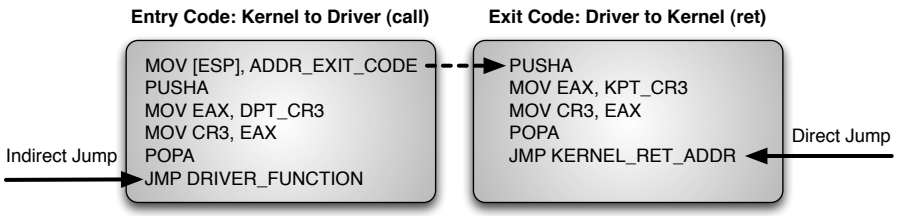


Figure 20: Runtime transition code generated by the kernel API monitor to enter in and exit from the driver code on indirect call and ret instructions, respectively.

API monitor copies the operand of the indirect call instruction from the kernel code over to the indirect jump instruction that it is generating on the transition page. For example, if an indirect call instruction is `ff d1`, the kernel API monitor generates the code `ff e1` on the transition page to jump to the driver function; that binary code is the equivalent indirect jump with the same operand. The kernel API monitor also generates the exit code to return control from the driver back to the kernel. The exit code for an indirect call is identical to the exit code for a direct call instruction with a hardcoded jump location. Figure 20 shows the transition code that the kernel API monitor generates for an indirect call instruction from the kernel to a driver and for its return.

5.4.2 Dynamic In-Memory Code Rewriting

The kernel API monitor redirects calls through the transition pages by dynamically rewriting the call instructions on the code pages of the kernel and drivers when those calls cause transitions between the DPT and KPT. In combination with the runtime code generation, this redirection allows address space switching to occur at native speed. The on-demand rewriting allows a `call` instruction to fault once. During the processing of the fault, the kernel API monitor first validates the control flow transfer. If it finds the transition valid, it generates transition code and rewrites the faulted call instruction to point to the entry code. The kernel API monitor also removes the fault caused due to the `ret` instruction. This design does not let the execution fault

on the verified re-written instructions for every future invocation: this call and its return are now on the fast path.

We first describe the on-demand dynamic binary rewriting of direct `call` instructions that transfer control from drivers to the kernel. A direct `call` instruction contains one byte of opcode and four bytes of operand specifying the location of the invoked kernel function. On a fault, the kernel API monitor rewrites this call instruction by replacing its operand with the address of the entry code generated on the transition page; the original target operand is inserted on the transition page as the jump target of the entry code. With this rewriting, the existing direct call instruction to the kernel function becomes the direct call instruction to a transition code present on the transition page.

The kernel API monitor also rewrites indirect call instructions in the core kernel targeting drivers. (Note that the kernel never targets a loadable driver with a direct call as such kernel code would fail to statically link.) Overwriting indirect call instructions with direct calls is complicated because most x86 indirect calls are 2 bytes, 3 bytes, or 6 bytes in length. The kernel API monitor needs 5 bytes to rewrite an indirect call instruction with a direct call instruction targeting transition code.

To perform the rewriting of short indirect call instructions, we insert `NOP` instructions in the kernel binary after each indirect call instruction at kernel compile time. An indirect call instruction followed by `NOP` padding provides sufficient width to replace the indirect instruction with the direct call. We modify `gcc` so that it generates new binaries containing `NOP` instructions after each indirect call instruction. With the new kernel binary, the kernel API monitor is able to overwrite indirect call instructions in the kernel code with direct call instructions pointing at entry code on the transition page.

Importantly, note that *our design does not require drivers to be recompiled* with the modified compiler. We specifically chose this design because it does not force

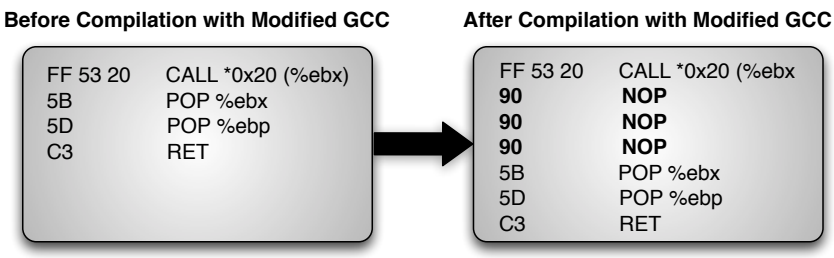


Figure 21: Effect of compilation of the kernel with the modified GCC that adds nop instructions after each indirect calls.

third party vendors (or full-kernel malware authors) to compile their drivers with our compiler; it is also one of the reasons why we do not rewrite indirect calls from drivers to the core kernel. A second reason to not rewrite indirect calls from drivers to the kernel is security. Recall that during the code generation of indirect call instructions, we copy the operand of the call instruction to our transition page. An attacker could easily change the value stored in the registers that are part of the operand and could invoke unchecked arbitrary kernel functionality. Our design does not allow such transitions into the core kernel and strictly enforces the non-bypassable interface to drivers. Figure 21 shows a snippet of the kernel's binary code and its transformation after compiling it with the modified gcc.

5.5 Alternative Design for Windows Operating Systems

Currently, the kernel API monitor isolates all drivers into a single address space separate from the kernel. An alternative design could use provenance information associated with drivers to achieve more flexible isolation. For example, it could position drivers signed by trusted parties, such as Microsoft, together with the kernel code in the KPT. Only drivers whose provenance is either not known or not verified would be isolated in the DPT. This isolation strategy would further reduce the overhead of the kernel API monitor because operations involving trusted drivers in the KPT execute at full speed without inter-positioning costs; only drivers in the DPT require binary

rewriting and code generation.

The alternative design would enable the kernel API monitor to monitor an untrusted driver's interaction with both the core kernel and the KPT drivers. Given that users install drivers from different third party vendors without knowing their provenance, an inflexible preventive approach that outright blocks the loading of new drivers may not work in practice. Our flexible design, in contrast, could be adopted by commodity operating systems vendors such as Microsoft to restrict the operation of untrusted modules or drivers. The provenance based design further limits return-oriented programming [63], previously discussed in Section 5.3.1, as code from neither the core kernel nor any trusted driver could be used for gadget construction.

The kernel API monitor differentiates between trusted and untrusted drivers at the time of loading to decide in which address space they must be mapped. This differentiation can be made using certificates. For example, a driver signed by Microsoft can be loaded in the trusted address space. However, Microsoft might not rely on drivers signed by other parties whose authenticity is not verified. With this design, all Microsoft signed drivers are loaded into the trusted address space, and other drivers signed by third party vendors or unsigned drivers, including kernel malware, are loaded into the untrusted address space.

The address space creation and switching techniques in Windows environment are similar to the one that are described in the Section 5.3.1 and Section 5.3.2. The notably difference is in the way trusted and untrusted code pages are identified. In Windows environment, existing kernel space, called the *trusted page table* (`TPT`), contains all the core kernel and trusted driver code with read and execute permissions, and untrusted driver code with read-only permissions. The untrusted driver address space, called the *untrusted page table* (`UPT`), contains untrusted code with read and execute permissions, and trusted code as non-readable, non-writable, and non-executable. The kernel API monitor also makes sure that the data pages mapped

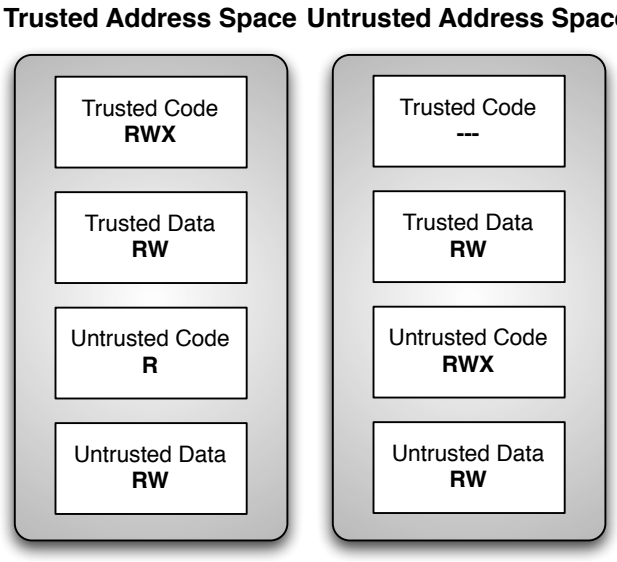


Figure 22: Layout of kernel and driver address spaces with permissions set on memory pages.

in both the address spaces are non-executable. Figure 22 shows the permissions set on UPT and TPT memory pages. With these permission bits, any control flow transfers from untrusted to trusted address space induce page faults thereby enabling the host-attribution software to monitor kernel APIs invoked by untrusted drivers.

To record parasitic attacks as described in Cases 2A & 2B, our monitor uses an automaton to characterize the parasitic behavior originating from malicious drivers. When the monitor intercepts kernel APIs, it verifies against the automaton to recognize the parasitic behavior. In our current prototype, we create an automaton based on the kernel APC-based code injection (Figure 23). To detect attacks as described in Case 2C, we determine the actual malicious driver by enumerating all threads of a process and match against threads of untrusted drivers.

5.6 Kernel API Monitoring

The kernel API monitor records all kernel functions invoked by drivers through the non-bypassable interface. Our optimized design creates two different paths by which

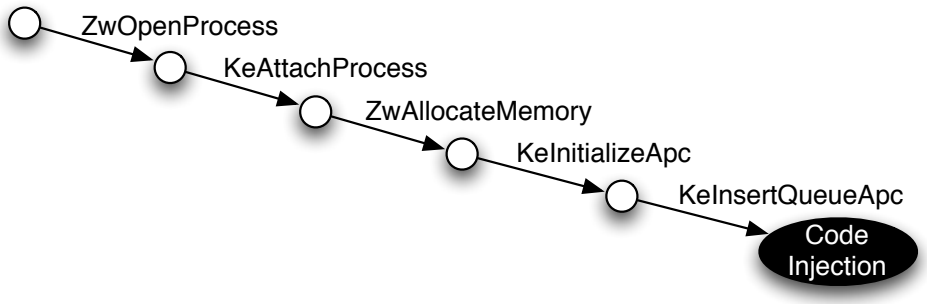


Figure 23: Runtime parasitic behavioral model for a driver-to-process injection.

drivers may invoke kernel functions: a slow path that causes a world switch and a fast path that uses transition pages. To be able to record all kernel functions invoked by drivers, our software must monitor both the slow and the fast path.

To monitor kernel API invocations on the slow path, the monitor records the virtual address of the invoked kernel function at each page fault from drivers to the core kernel. It also finds the virtual address of the callee by using the return address present on the stack. Since the kernel API monitor protects the return address, it can securely identify the driver that invoked the kernel function. Once the monitor identifies the source and destination information, it records this data to be used by higher-level security software.

Monitoring API invocations via the fast path requires a different strategy. Since fast path memory barrier transitions do not reach the hypervisor, the kernel API monitor will not be able to record these APIs invocations. To be able to monitor the kernel APIs on the fast path, our software augments the code generated on transition pages so that it additionally logs the kernel API invocation information in protected guest kernel memory. Our software allocates guest memory pages called *log memory* from the hypervisor and uses this memory pool to store the API invocation information occurring through the fast path. In a standard producer-consumer model, the logs are written inside the guest by the transition code, and the kernel API monitor

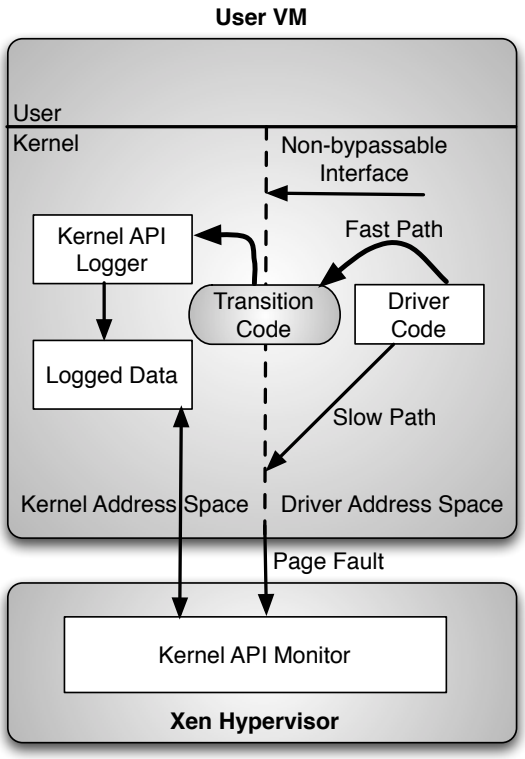


Figure 24: Low-level architecture of the Kernel API monitor that records kernel APIs on both the slow and the fast path.

reads the logged data asynchronously from the hypervisor. To protect the logged data from malicious drivers, we mark the allocated memory pages as non-readable, non-writable, and non-executable in the DPT. These pages have read-write permissions inside the KPT. With this design, the monitor is able to log all kernel API invoked by drivers both on the slow and fast paths. Though our fast path design is secure, and it improves the performance, it introduces delay in detecting attacks because all APIs are logged and periodically read by the monitor. Figure 24 shows the architecture of the kernel API monitor.

To record the API information in the guest memory, the kernel API monitor first generates a logging function called `Klog` on a separate guest kernel memory page mapped as non-readable, non-writable, and non-executable in the DPT. It is read-only and executable inside the KPT. Then, during the generation of the entry code on

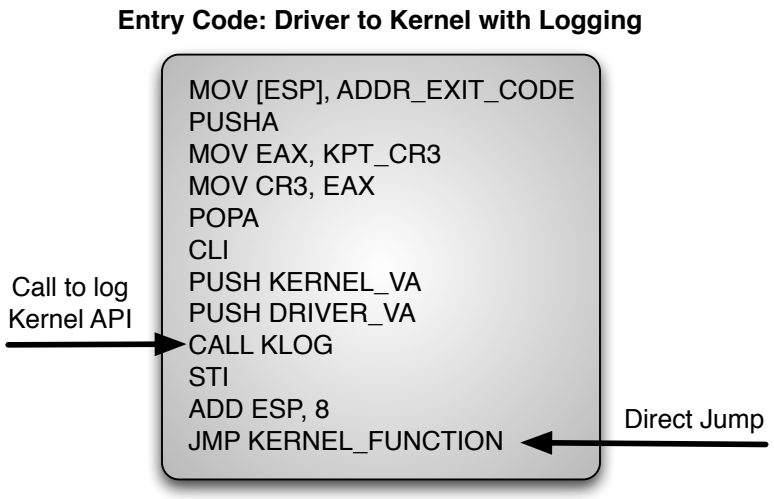


Figure 25: Runtime entry code generated by the kernel API monitor to enter into the kernel code from drivers. This code includes the API logging logic also.

a transition page, we add extra instructions that invoke Klog from the transition page after switching CR3 to the KPT. When a direct call from drivers to the kernel goes through the transition page, the transition code invokes the Klog code. The transition code also passes the virtual addresses of the called and callee functions using the kernel stack. Figure 25 describes the transition code augmented with the logging code. To avoid attacks in which untrusted drivers tamper with the information present on the stack, we first switch the CR3 from the `DPT_CR3` to the `KPT_CR3` and then push the information on the stack. In this design, driver code becomes non-executable and Klog extracts the information in a secure way. To avoid security issues due to interrupts during logging, we disable interrupts before parameters are pushed on the stack and enable them after Klog completes. Klog writes invocation information in the log memory to be consumed by the hypervisor, which subsequently passes the data to any high-level security software.

5.7 Security Evaluation

We evaluated our monitor's enforcement of the non-bypassable interface and monitoring of the kernel APIs invoked by drivers.

5.7.1 Non-Bypassable Interface Evaluation

We tested the kernel API monitor's ability to enforce the boundary between drivers and the kernel code with a synthetic malware instance that jumps into the middle of the kernel code to execute an operation. When we ran our malicious driver inside the guest VM, the kernel API monitor loaded it into the DPT and marked its code pages non-executable and non-writable in the KPT. When the malicious driver tried running its malicious code from its initialization function, the `jmp` instruction caused a fault into the hypervisor as the kernel code was not present in the DPT. On verification, our system correctly found that the target address was not a valid kernel entry point and raised an alarm.

5.7.2 Kernel API Monitoring Evaluation

We evaluated our system's ability to monitor all kernel APIs invoked by malicious drivers. We ran the kernel API monitor with two malicious drivers: the `lvtes` keylogger and a synthetic kernel-mode bot. The keylogger installs a kernel driver, receives user keystrokes, and logs them to a file. It performs all operations inside the kernel, and it does not contain any user-space process. When we loaded lvtes in the guest VM, the kernel API monitor set up the appropriate permissions for the code and data pages of lvtes both in the KPT and DPT. During its execution, lvtes invoked several kernel APIs to read data, to write data to the log file, to hide the file, to allocate memory, and to perform some other functions. Our monitor was able to log all APIs invoked by lvtes, shown in Table 12.

In our second test, we ran the kernel API monitor with a synthetic kernel-level bot having basic functionalities, such as socket creation, network connection, data

Table 12: Kernel APIs invoked by the lvtes keylogger and the kernel-level bot.

Malicious Driver	*Kernel API Invoked*
Lvtes	sys_open, sys_read, sys_write, sys_close sys_getdents, _spin_unlock snprintf, __wake_up, __kmalloc copy_from_user, copy_to_user, _spin_lock, kfree, memmove
Full Kernel Bot	kmem_cache_alloc, sock_create, inet_stream_connect sock_recvmsg, sys_sendmsg

transmission, and packet receipt. This bot again completely resided in the kernel as a driver, and it did not have any user-level component. We ran one server on a separate test machine so that bot could communicate with it. After loading the bot and isolating its code pages in the DPT, when it executed its functionality, the kernel API monitor successfully detected all API functions invoked by the bot, listed in Table 12.

The above results show that the kernel API monitor is effective in enforcing the non-bypassable interface. Given this interface, our monitor then logs all interaction of drivers with the core kernel. The information provided by our monitor can be used by high-level security software that could, for instance, identify malicious software based on their unusual use of the kernel interface. To demonstrate the usefulness of the kernel API monitoring, we empirically evaluated the difference between the kernel APIs invoked by malicious and legitimate drivers. Table 13 shows the key kernel APIs invoked by file system and networking drivers on our test system. A comparison of the two tables shows a clear distinction between the set of APIs invoked by malware and legitimate drivers. These anomalies can be used by high-level security software to detect attacks.

Table 13: Kernel APIs invoked by the benign drivers and logged by the kernel API monitor.

Benign Drivers	*Kernel API Invoked*
File system driver	kmem_cache_alloc, clear_inode, new_inode, generic_commit_write, block_prepare_write, block_write_full_page, generic_file_aio_write, rb_erase, _spin_lock, _spin_unlock, truncate_inode_pages, submit_bh, rb_first
Network driver	_spin_unlock, __alloc_skb, eth_type_trans, _spin_lock, netpoll_trap, raise_softirq_ireoff, _spin_lock_irq, _spin_unlock_irq, netif_receive_skb

5.7.3 Kernel-level Parasitism Identification

We evaluated the kernel API monitor's ability to detect kernel-level parasitism by testing it with the recent storm worm [104]. Storm is kernel-level malware that exhibits parasitic behaviors by injecting malicious DLLs into the benign `services.exe` process, causing `services.exe` to launch DDoS attacks. We loaded storm's malicious driver in the test VM. Since the driver is untrusted, the kernel API monitor loaded it into the separate isolated address space. On the execution of the driver's code, all kernel APIs invoked by the driver were verified and logged by the kernel API monitor. The monitor found that the driver was performing injection via APCs, and it recorded both the parasitic behavior and the victim process.

5.8 Performance Evaluation

This section measures the performance overhead incurred by our monitor on CPU, network, disk, and application benchmarks.

5.8.1 Compatibility Evaluation

We designed and developed the kernel API monitor to offer its protection to the kernel from drivers. We conducted a compatibility test to show that the kernel API monitor

Table 14: Statistics related to the kernel API monitor's implementation and impact on a running system.

Task	*Count*
Lines of gcc source code modified	5
Drivers isolated	36
Approximate direct instructions overwritten	500
Approximate indirect instructions overwritten	65
Approximate transition pages used	8

did not make any assumption on driver code. We tested the kernel API monitor with 36 commodity Linux drivers, and the kernel API monitor was able to isolate all of them. Further, the kernel API monitor was able to perform binary rewriting and runtime code generation for all these drivers and the core kernel. Table 14 presents detailed statistics related to the kernel API monitor's implementation and basic impact on the guest kernel. Our compatibility evaluation shows that the kernel API monitor's design is effective, and it can be used to protect operating systems from drivers, including kernel-malware. Table 15 shows the list of all isolated commodity drivers along with their sizes.

5.8.2 Experimental Evaluation

We evaluated the kernel API monitor's impact on a system's performance with extensive benchmark-driven evaluation. Our testbed contained an Intel 2.8 GHz Core 2 Quad processor, 4 GB of RAM, and a 100Mbps ethernet card. We used the Xen hypervisor in PAE mode, and our guest user VM used the 32-bit Linux 2.6 kernel. For our experiments, we assigned 1GB of memory to the guest VM, and 3 GB of memory was shared between the security VM and the hypervisor.

We tested the kernel API monitor with a collection of benchmarks exercising the CPU, disk I/O, and network I/O: Lmbench [78], BYTEmark [152], Iperf [138], and Bonnie [150]. We performed all experiments five times and report median values.

Table 15: List of commodity Linux drivers that the kernel API monitor isolated in the DPT during system evaluation. No driver execution resulted in the kernel API monitor alerts or control flow failures.

Module	*Size*
ppdev	9220
autofs4	20100
hidp	16640
l2cap	25088
bluetooth	46308
sunrpc	141884
ip_conntrack_netbios_ns	3328
ipt_REJECT	5632
xt_state	2432
ip_conntrack	50860
nfnetlink	6808
xt_tcpudp	3456
iptable_filter	3328
ip_tables	12232
x_tables	13060
video	15876
button	7056
battery	9732
ac	5252
ipv6	235744
lp	12872
parport_pc	26276
parport	36040
floppy	60100
nvram	9096
i2c_piix4	8848
i2c_core	21120
8139too	26752
8139cp	21888
mii	5632
dm_snapshot	16428
dm_zero	2048
dm_mirror	20432
dm_mod	51992
ext3	121864
jbd	55700

Table 16: Execution time measured by lmbench without and with the kernel API monitor for context-switching, procedure calls, and system calls. Times reported in nanoseconds; smaller measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

Operation	*Normal VM (ns)*	*The kernel API monitor VM (ns)*	*Overhead (%)*
Context-switch	2,400	2,560	6.67
Procedure call	3.6	3.6	0
System call	80.6	80.7	0.12

Table 17: Network latency and throughput measured by lmbench and lperf without and with the kernel API monitor. Smaller measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

Operation	*Normal VM*	*The kernel API monitor VM*	*Overhead (%)*
TCP latency (μs)	431.6036	470.9497	9.12
UDP latency (μs)	432.0758	454.3676	5.16
Connection latency (μs)	908.0500	966.9074	6.48
TCP throughput (MB/sec)	8.08	7.98	1.23
UDP throughput (MB/sec)	1.06	1.06	0

We present the results of file-system benchmarks in boxplots due to high variance in disk I/O measurements. In our results, "Normal" refers to measurements that do not have the kernel API monitor's protection and "The kernel API monitor" includes our protection.

In our micro-benchmark experiments, we first measured the effect of the kernel API monitor on operations that happen very frequently. Using lmbench, we measured the cost of a context-switch, procedure call, and system call. Table 16 shows our results. It can be seen from the table that the kernel API monitor's overhead on the regular operations is low.

In another experiment, we measured the kernel API monitor's overhead on network operations. Since the kernel API monitor isolates all drivers—including the networking driver—in another address space, we measured this effect. We connected

Table 18: The kernel API monitor's overhead on CPU-bound applications as measured with BYTEmark; higher measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

Operation	*Normal VM (iteration/sec)*	*The kernel API monitor VM (iteration/sec)*	*Overhead (%)*
Numeric sort	1095.80	1092.20	0.33
String sort	163.45	162.80	0.40
FP emulation	186.28	185.61	0.36
Fourier	30498	30390	0.35
Assignment	37.63	37.37	0.69
Idea	5806.70	5786.70	0.34
Huffman	2358.10	2347.80	0.44
Neural net	45.52	45.45	0.15

two machines with a switch. We used lmbench's network tests to measure network latency, TCP and UDP latencies, and throughput of TCP connections, and Iperf to measure UDP throughput. The results are shown in Table 17. Although the kernel API monitor's overhead on network operations is low, it still affects TCP communication. We investigated the cost of TCP operations, and we found that some of the functions on the TCP code path were not receiving fast path optimization because the driver was invoking kernel functionality via indirect calls. Since the kernel API monitor does not rewrite indirect instructions from drivers to the kernel, these control flows remain on the slow path.

To measure the kernel API monitor's effect on CPU-bound execution, we tested it with computationally intensive work loads. We performed these experiments with BYTEmark, a benchmark that runs various CPU-intensive algorithms and measures the performance in iterations per second. We tested the kernel API monitor with all tests, and Table 18 shows our results. They indicate that the kernel API monitor's overhead on CPU-bound applications are very low when compared with normal execution.

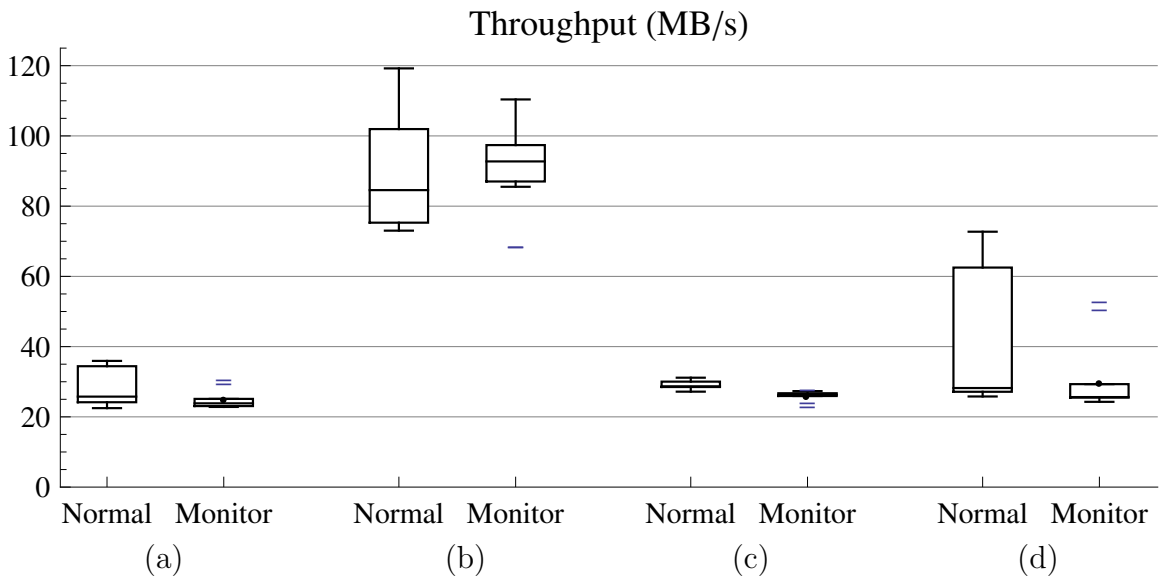


Figure 26: The kernel API monitor's impact on the filesystem measured with Bonnie. All measurements show throughput in MB/s; higher measurements are better. Here, "Normal" refers to measurements that do not have our protection and "Monitor" includes our protection. Boxes show medians and first and third quartiles. Outliers appear as dashes. Groupings show performance of (a) character reads, (b) block reads, (c) character writes, and (d) block writes.

We next measured the effect of the kernel API monitor on file system performance. Since we isolated the file system drivers in the DPT, we measured the effect of this partitioning. We carried out this experiment with bonnie, a benchmark that measures the throughput of read and write operations performed in both character and block sizes. In this experiment, we created a file of size 2 GB, which exceeds the size of the memory allocated to guest VM to reduce caching effects. Our results, shown in Figure 26, indicate that the kernel API monitor's read and write operations' throughput remains close to the normal VM's results.

5.8.3 Effect of Fast Path Optimization

Previous experiments showed that the kernel API monitor's overhead on the fast path was low. In this set of experiments, we specifically compared the performance of the fast path with the slow path implementation of the kernel API monitor to

Table 19: Effects of the kernel API monitor's fast path design; smaller measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

Operation	*Normal VM (sec)*	*Slow Path (sec)*	*Overhead (%)*	*Fast Path (sec)*	*Overhead (%)*
make	64.055	80.297	25.37	67.338	5.12
bzip2	41.847	51.777	23.73	43.287	3.44
tar	29.434	40.511	37.63	30.109	2.29

show the effect of fast path design. Our test included a compilation of the stripped-down version of the Linux kernel, file compression, and tarring of the Linux source directory. Our results, presented in Table 19, show that the kernel API monitor's fast path design has improved the system's performance substantially when compared to the overhead on the slow path. These results also confirm that our design of fast path is efficient, and the overhead of the kernel API monitor is acceptable on the system.

5.8.4 False Positive Evaluation

We tested the kernel API monitor's proclivity to falsely block legitimate driver behavior by loading and using benign device drivers in the presence of our tool. A false positive occurs in the kernel API monitor if benign drivers bypass the kernel's exported interfaces and execute control transfers to internal kernel code. We analyzed 36 benign drivers loaded into our test guest VM and found that none made invalid control transfers.

5.9 Conclusions

The kernel API monitor records the interaction of drivers with the core kernel by creating a non-bypassable interface inside the kernel. It isolated all drivers from the kernel code by creating a separate address space for drivers. The address space isolation incurred performance overhead because each address space switch caused world

switches to the hypervisor. The kernel API monitor solved this problem by establishing a fast path, which used on-demand runtime binary rewriting of guest kernel and code generation. With this design, the kernel API monitor allowed most control flow transfers between drivers and the core kernel to occur at native speed. The creation of an efficient, non-bypassable interface allowed the kernel API monitor to monitor kernel APIs invoked by driver through the interface. Our evaluation showed that the kernel API monitor's interface enforcement was effective, its monitoring was capable of logging kernel APIs and detecting kernel-level parasitic attacks, and its overhead on the system was low.

CHAPTER VI

KERNEL DATA PROTECTION

6.1 Motivation

The process attribution software, described in Chapter 3, relies on the monitored VM's kernel data to correctly correlate malicious network traffic to host-level processes. As described in Section 3.6.4, kernel-level malware instances can tamper with the critical kernel data to evade the process attribution software. Kernel-level malware often uses a technique called direct kernel object manipulation (DKOM) [146] to hide the existence of malicious processes by eliding task structures for the processes from the kernel's process accounting list. Malicious kernel-level components can hide their own presence by illicitly removing data structures identifying their presence from a kernel-managed list of loaded drivers or modules. They may elevate a process' privileges by overwriting the process' credentials with those for a root or administrative user. The process attribution software relying upon the core kernel's own management information will fail to identify the presence of malicious processes.

The key contribution of this chapter is to present a kernel data protection approach that creates access control protections for security-critical kernel data. The kernel data protection problem is challenging due to the unified kernel memory space, which allows a malicious kernel component to alter the core kernel's private data at will. Our approach solves this problem by partitioning kernel memory into separate regions having different access control policies or restrictions detailing when the code of the core kernel and its loaded components can access sensitive data in a particular protected region. The hypervisor mediates the execution of instructions attempting to write protected sensitive kernel data. All other accesses to non-sensitive data occur

without any mediation. Our data protection system enforces access control policies that specify what code regions of the kernel are allowed to write what data objects within kernel memory.

6.2 Previous Approaches

Our primary contribution is integrity protection for kernel data. Previous studies have examined aspects of this problem and arrived at solutions different than our own. Petroni et al. [114] proposed a system that detects semantic integrity violations in kernel objects, such as a process task structure reachable when traversing a linked list used by the scheduler but not reachable when traversing the process accounting list. Baliga et al. [7] improved Petroni's work by developing an automated system to generate invariants on kernel data structures. Periodic invariant verification attempts to discover the sort of data manipulation addressed by our work, but it has some limitations overcome by our protection software. These techniques succeed only when invariants can be stated for a data object. This is clearly possible for structures like a process list, as the invariant can compare the reachable nodes along two different traversals: process accounting list and process scheduling list. It is not evident that invariants can be written for other structures, such as the list of loaded kernel modules, which do not offer multiple views. Our data protection software operates differently: it mediates all attempted data alterations and allows only those invoked by legitimate kernel functionality. While previous approaches detect malicious modifications, we prevent the illegitimate changes of critical data from even occurring.

XFI [32] and BGI [14] provide integrity protection for data (among other protections) via guarded write instructions in software components subject to access control policy constraints. Their use of inlined verification imposes restrictions on software development that may prove difficult to satisfy in actual deployments due to the presence of malicious drivers. For example, they require buy-in from all kernel drivers

and modules (including rootkit modules). In contrast, the kernel data protection software operates with only cooperation from the core, static kernel; dynamically-loaded components are unconstrained. The designs of XFI and our system also highlight differences between inlined monitoring and external protection. XFI guards all computed writes to ensure that no write kills a protected value. When many of these writes target unprotected addresses, performance still degrades. The kernel data protector, in comparison, mediates writes only when they actually attempt modification of protected data. XFI's protections occur at the origin (every write instruction), whereas our protections occur at the destination (the security-critical data).

We partition kernel memory and data objects into protected and unprotected regions. The general concept of partitioning an object into secure and insecure portions has appeared as a solution to other software security and reliability problems. Multics' protection rings created different memory regions having different access permissions [24]. The Pentium architecture created hardware based rings to isolate user-level processes and kernel-level services [65]. Mondrian [163], a fine-grained memory protection system, allows multiple protection domains to share and export services. Ta-min et al. [148] proposed a system that partitions the system-call interface into secure and insecure components. A hypervisor processes secure system calls while insecure calls are handled as usual. Xiong et al. [165] proposed a kernel integrity protection system that isolates untrusted drivers in a separate address space and monitors accesses performed by them. They relied on a driver signing based approach to distinguish between benign and untrusted drivers. Further, their system also protected kernel data, but they did not employ any object partitioning approach like our system, hence they incurred up to 21% overhead. Payne et al. [113] split a single security application among multiple VMs and protected the components placed in any untrusted VM. Chong et al. [19] sub-divided web applications to ensure that the resulting placement of code and data are secure and efficient. Brumley et al. [13]

automatically partitioned a single program into a privileged program that handles all the privileged operations and an unprivileged program that does everything else. Like these examples of interface and application partitioning, the goal of our system's memory partitioning is to improve kernel security. However, unlike previous systems, the purpose of our object partitioning is to improve the performance of the protected kernel.

6.3 Overview

A running kernel aggregates code and data from the core kernel and from a collection of dynamically-loaded modules and drivers. These different components may engender varying levels of trust in regions of the kernel. We assume that the core kernel implemented by the kernel developers receives full trust. Many modules and drivers, however, have unknown provenance and hold only limited trust. Our protection software monitors the interactions between code with low trust and critical data with high trust.

The core kernel includes operations and data exported to modules for their use as well as internal functionality and objects meant to be managed only by the core kernel itself. For example, the list of loaded modules, the process accounting list, the scheduler list, and the run queue of the Linux kernel exist in the core kernel's data and heap space and are altered by internal functionality of the kernel. However, the lack of memory barriers between the core kernel and its dynamically-loaded components prevents the kernel from disallowing illicit alterations of its internal data structures by malicious modules or drivers.

Consider as an example the Linux kernel's task management. The kernel stores per-process data, such as user IDs and group IDs that determine a process' privilege level and allowed access, in an aggregate data structure called the `task_struct`. Each `task_struct` also contains references to the next and previous task structures and thus

```
asmlinkage int give_root()
{
  if(current->uid != 0)
  {
    current->uid = 0;
    current->gid = 0;
    current->euid = 0;
    current->egid = 0;
  }
  return 0;
}
```

Figure 27: Fragment of rootkit code that elevates privileges of non-superuser processes to superuser (ID 0).

```
int init_module()
{
  task = find_task_by_pid(pid);
  if (task) {
    REMOVE_LINKS(task);
  }
}
```

Figure 28: Fragment of rootkit code that removes a malicious process identified by `pid` from the process accounting linked list.

forms a node in a doubly-linked list. Security tools in the kernel or in a monitoring virtual machine find the set of running processes by traversing the doubly-linked list [67, 140].

Suppose that a malicious application wants to execute undesirable functionality as a high-privilege user while remaining elusive from security software that searches for unexpected processes. The application may include a kernel-mode rootkit that elevates the malicious process' privilege by directly writing to its `task_struct` and hides the process by altering the kernel's process accounting list. Figure 27 shows a C code snippet from a rootkit [107] that assigns superuser privileges to its malicious process. In the code shown in Figure 28, a rootkit [151] uses the kernel macro `REMOVE_LINKS` to remove its malicious process' `task_struct` from the doubly-linked process accounting list. This macro alters the `next` and `previous` pointers within the list and allows the rootkit to hide its malicious process.

6.3.1 Kernel Data Integrity Model

Potentially malicious kernel modules should not be able to alter security-critical data intended to be managed only by the core kernel. We protect such critical data and enforce data access restrictions based upon the origin of the access within the code of the kernel and its modules or drivers. The data integrity model is straightforward and matches that of the Biba ring policy [9]:

- Trusted core kernel code may write to any kernel data.
- Loaded modules and drivers can write to any low-integrity data, which includes all driver data and portions of the core kernel data.
- Loaded modules and drivers cannot write to high-integrity data of the core kernel either through a direct write or by calling into existing code of the core kernel that will execute the write on behalf of the module, unless the control-flow transfer into the core kernel targets an exported function. Most exported functions act as *trusted upgraders*. The intent of the kernel developers was to provide APIs through which modules and drivers can legitimately make changes to critical data, and the changes leave the kernel in a consistent state.
- Exported library calls that alter raw memory, such as memcpy, memset, or bcopy, are not trusted upgraders. Since these functions can arbitrarily manipulate raw memory, changes made directly by these APIs may leave critical kernel data in an inconsistent state.

To determine if a loadable kernel component called into a core kernel function to induce alteration of high-integrity data, we perform a stackwalk of the kernel stack to identify activation records and the call chain that led to an attempted write. The use of stack inspection to determine access control is similar to previous approaches used in the context of Java [33, 158].

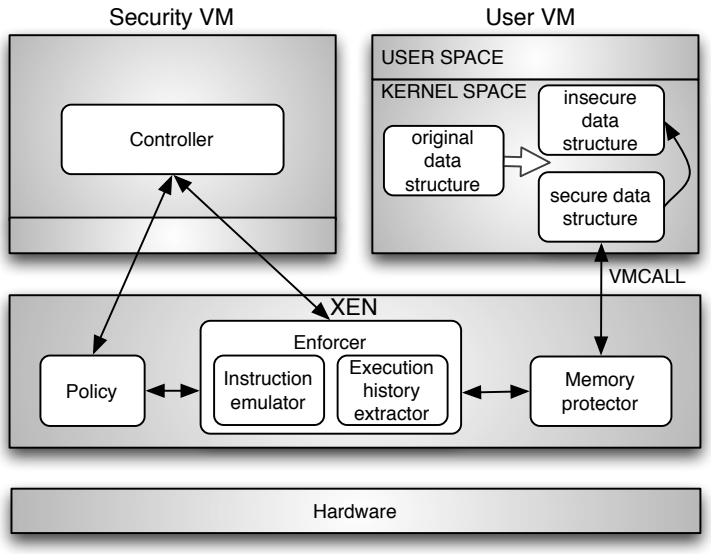


Figure 29: The architecture of our kernel data protection software.

Consider again the rootkit behaviors of Figures 27 and 28. When the rootkit attempts to directly modify the privileges of a process, our protection software will mediate the write to security-critical data of the core kernel. The malicious code that modifies privileges by directly writing to memory is in a loaded module and not in the core kernel code, so we will prevent the write (and optionally alert the system's user or administrator of a likely malware infection). Should the macro expansion of the rootkit's attempted unlinking of the `task_struct` from the linked list include function calls into internal list management functions of the kernel, then our stackwalk will step into an activation record for the code of the loaded module. Again, our kernel data protection software will deny the data write.

6.4 Kernel Memory Access Control

We design our kernel data protection software, shown in Figure 29, to protect sensitive kernel data from unauthorized modification. Its design reflects these goals:

- **Data structure protection:** Recent kernel malware instances hide their malicious activities by modifying dynamically allocated data structures using

DKOM. To thwart these attacks, we protect heap-allocated critical data structures.

- **Performance:** Our memory access control may incur high performance cost if designed naïvely. To keep the overhead low, we use memory partitioning to lay out sensitive data on separate memory pages and protects those pages using the hypervisor.

The hypervisor is the heart of our protections, comprised of the memory access policy description, a memory protection module, and a policy enforcement module. The automatically constructed policy statement includes a description of trusted code regions allowed to modify security-critical data. The memory protection module protects all kernel memory pages containing security-critical data. The policy enforcer mediates attempted writes to protected data and uses the policy to determine when writes should be permitted. If permitted, the enforcer's instruction emulator emulates the write operation in a manner transparent to the user VM. The enforcer includes functionality to extract execution history in the form of activation records present on the guest kernel's call stack.

6.4.1 Policy

We enforce integrity protection policies based on the trust level of code attempting to alter critical data. Our policies describe code regions or function call chains that are allowed to modify security-critical kernel data. Any access request that does not fall into pre-defined trusted code regions will be denied. We identify the following three types of code regions that can legitimately modify protected data:

1. All core kernel code (that not in loadable modules or drivers) is trusted to correctly manage its own private data. This code spans memory from the Linux kernel symbol `_text` to `_etext`.

2. Kernel code from `__init_begin` to `__init_end` contains code required for the kernel to successfully boot and is likewise trusted.

3. Alterations reachable from most exported kernel functions' entry points reflect valid management of private kernel data even when a loaded component calls into the function. Exported kernel functions are deliberate APIs, defined in Linux's System.map file, created by the core developers specifically for loadable modules and drivers. Excluding library calls that alter raw memory, these functions leave an operating system's kernel in a consistent state.

6.4.2 Activation of Mediated Access

To enforce integrity protections, we mediate all attempts to overwrite security-critical data. We use memory page access permissions to disable write permissions on any page holding sensitive kernel data. This protection provides our software with the ability to interpose on write accesses to protected memory pages; any write operation to a protected page causes a page fault, and on each fault the hypervisor gains the control of execution.

Our memory protection relies on knowledge of the location of dynamically-allocated sensitive kernel objects. It uses code instrumentation within the user VM's kernel to activate and deactivate protections in concert with object construction and destruction. The instrumentation uses the Intel hardware virtualization instruction `VMCALL` [65], which transfers control to the hypervisor when executed. In our design, each `VMCALL` passes the page frame number (PFN) and virtual address of the newly allocated memory page requiring protection. Since the core kernel code is write-protected, an attacker will not be able to remove our instrumentation. On receiving a VMCALL, we verify the location of the call, and if the location does not belong to the set of locations stored in the hypervisor, then it discards the call.

Our kernel data protection system receives the `VMCALL` within the hypervisor and

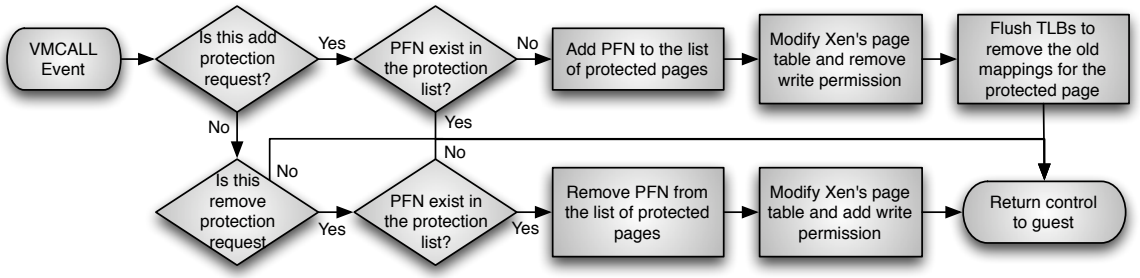


Figure 30: Steps involved during addition and removal of memory page protections.

handles the request from the guest. When the memory protection module receives a request to add protection for a guest's page, it adds the PFN to a list of protected pages and removes the page's write permission. We also flush the translation lookaside buffer (TLB) cache to remove any previous mappings that may exist for the protected PFN. When a request to remove protection on a page comes to the memory protector, it removes the PFN of the page from the list of protected pages and restores write permission. We thwart memory remapping attacks by keeping both virtual and physical addresses of protected pages in the hypervisor. Since any update to guest page tables will be synced to shadow page tables, we verify whether any protected virtual addresses have been remapped. If so, we additionally protect the new physical pages. Figure 30 presents a flowchart describing the entire process of adding and removing protection on a memory page.

6.4.3 Policy Enforcement

Our policy enforcer also resides in the hypervisor, and its duty is to enforce the predefined security policies. Both legitimate and malicious writes to protected pages will cause a page fault received by the hypervisor, providing opportunities for mediation. At each fault, the enforcer determines if it is because of our protection by verifying the PFN of the faulting page. If the PFN belongs to the list of protected PFNs, then it performs further actions. Otherwise, it directs the hypervisor to resume normal

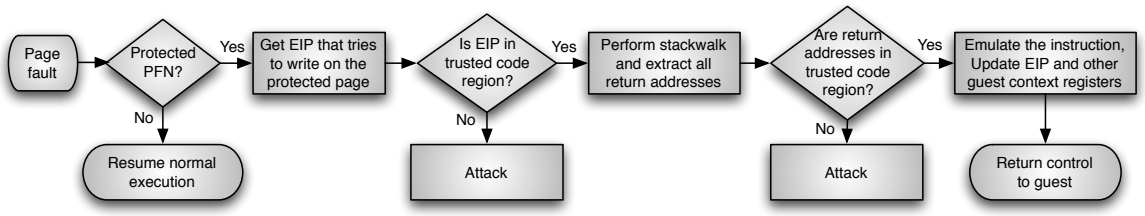


Figure 31: Steps used by kernel data protection software to resolve a write fault on a protected page.

operation. On a page fault caused by our protection, the enforcer first uses the user VM's instruction pointer (`eip`) to know which code is directly attempting to write to the protected page. If the instruction pointer belongs to an untrusted code region, then the access must be denied. If the instruction pointer belongs to a trusted code region, then the enforcer must ensure that the trusted code was invoked legitimately. It extracts the execution history of the kernel associated with the memory write by executing a stackwalk of the user VM kernel's stack. When encountering a stack frame for a core kernel function with a return address pointing back to untrusted code, our software checks to see if the core function is a trusted upgrader. If not, then untrusted code invoked an unsafe call into the kernel, and the memory alteration must be prevented. Otherwise, we allow the write operation.

Once we determine that a write is permitted by the integrity protection policy, it emulates the write in the same way that the guest system would have done had the protection been absent. When the emulation completes, our software updates the guest context registers so that the mediation of writes remains completely transparent to the guest. The complete flowchart of resolving a write page fault is shown in Figure 31.

6.4.4 Memory Layout Optimization

The layout of kernel objects in memory challenges our protection's ability to achieve the performance goal. Our policies protect individual kernel variables and fields of

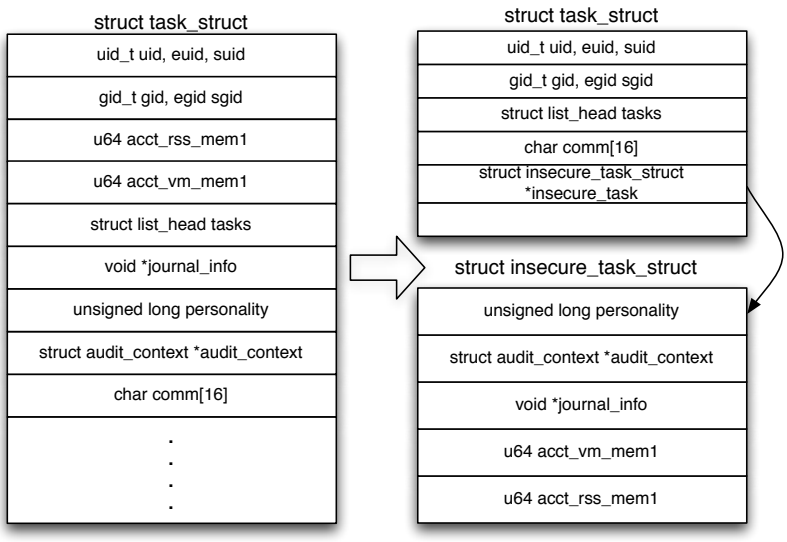


Figure 32: Memory partitioning of the Linux `task_struct` structure.

aggregate structures, but it mediates writes at the granularity of complete memory pages. Structures, like `task_struct` and `module`, contain a mix of security-critical and non-critical fields laid out contiguously in memory, thereby making it difficult to provide efficient and fine-grained protection. If any variable on a page is protected, the entire page suffers the overhead of mediated writes. We would prefer that only write operations to security-sensitive members invoke a fault, thereby eliminating the performance overhead resulting from faults generated by writes to non-critical data.

To address this problem, we develop two approaches to partition a data structure, say `Obj`, into secure and insecure pieces. All security-critical members (described in Section 6.4.5) of `Obj` should be located together on protected pages, and all non-critical members can reside on unprotected pages.

6.4.4.1 Structure Division

The first approach, *structure division*, partitions `Obj` by creating a new data structure `insecure_Obj` and moving all non-sensitive members into this new structure. For example, Figure 32 shows the partial division of the Linux `task_struct` into two structures: `task_struct` (secure) and `insecure_task_struct` (insecure). This

procedure leaves the original `Obj` structure with security-critical members only. A kernel using partitioned structures must allocate memory separately for the secure and insecure pieces to create different memory regions for protected and unprotected members.

To use partitioned structures, existing kernel code must be updated. All members that belong to `Obj` can still be accessed as before. However, code that accesses non-critical members must access them through `insecure_Obj`. To solve this problem, we first link `Obj` and `insecure_Obj` by adding a new pointer field in `Obj` called `insecure` that points to the `insecure_Obj` structure. Second, we modified all affected kernel code by adding an indirection through the `insecure` pointer. For example, if code in the kernel was accessing the field `journal_info` as `current->journal_info`, it is modified to become `current->insecure->journal_info`, where `current` points to the `task_struct` of the currently executing process. Although this strategy requires code changes, retrofitting the existing kernel code to use a new structure can be automated using source-to-source transformation techniques, such as those provided by CIL [100]. Moreover, these are one-time changes and subsequent, unrelated kernel development may incur only small maintenance costs as developers choose to shift variables to or from the protected region.

6.4.4.2 Structure Alignment

Our second partitioning approach, *structure alignment*, places security-sensitive and non-sensitive variables of a unified data structure on separate memory pages by aligning the structure in memory so that it lays across a page boundary. Only one of the two pages is protected by our system, and the structure's security-critical fields lay on that page. To accomplish the structure alignment, we group all security-critical members in `Obj` together at the start of the structure, group all non-critical fields at the end, and add a new alignment buffer to the structure between the two fields.

This buffer is simply padding that forces all non-critical members onto the second, unprotected memory page.

These partitioning strategies have trade-offs. Structure division provides freedom in laying out the protected and unprotected structures in memory at runtime, though it needs kernel source code revision. Structure alignment has only a minimal source code change to the definition of the structure, and it could easily be integrated into an existing kernel by creating a compile-time option that inserts or removes the alignment buffer. However, it may not be an effective use of kernel memory, and it imposes constraints on runtime memory layout. We demonstrate in Section 6.5 that both designs are feasible, even on pervasive kernel data structures.

6.4.5 Identifying Security-Critical Members

Our memory layout optimization design depends upon the identification of security-critical and non-critical members in a data structure. We define a member in a data structure to be security-critical if:

- It is manipulated by attackers to carry out malicious activities. This approach provided a reasonable idea of those data structure fields commonly modified by attackers and needing immediate protection. For example, many Linux-based rootkits modify uid and gid fields in a `task_struct` to elevate privileges of their malicious process. In another example, they hide the presence of their malicious modules by modifying the `next` and `prev` pointers of a `module` structure. Based on this notion, we collected rootkits to identify kernel variables that they alter.

- Subject-matter experts, such as core kernel developers, identify the variable as security critical. They can identify important members in a kernel data structure during a development phase before they are misused by attackers.

- Defensive systems rely on the variable's integrity in order to understand the security state of the user VM. For example, VMI applications such as the process

attribution software [67,140] rely on the integrity of process accounting lists and filesystem structures when constructing a trusted view of the system.

We believe that these criteria offer sufficient ability to identify security-sensitive members present in kernel data structures. Alternatively, operating system developers could identify low privilege data that is allowed to be modified by all drivers, and all other data could be protected.

6.5 Implementation

We developed our kernel data protection software using Linux 2.6 guest operating systems and the Xen hypervisor version 3.2. We describe low-level implementation details of our system in this section.

6.5.1 Data Structure Layout

We applied our partitioning strategies to two important Linux kernel data structures: `task_struct` and `module`. We chose these two data structures due to their complexity, their relevance to current kernel-level attacks, and their pervasiveness in the kernel. The process data structure is important to the kernel because it is the fundamental unit of execution, and its complexity is based on the fact that it contains 122 members. The module data structure has 29 members, and it is used when any driver or module is loaded or unloaded, and whenever any subsystem of the kernel implemented as a driver, such as a filesystem, disk, or network device, is accessed. To demonstrate the feasibility of our two partitioning strategies, we partitioned each of the two structure types with different techniques. We applied structure division to the widely-used `task_struct` and structure alignment to the `module` structure.

6.5.1.1 Partitioning of `task_struct`

To apply our partitioning strategy to the `task_struct` structure, we first identified its security-critical members. As described in Section 6.4.5, we identified critical

members by analyzing rootkits and the Linux kernel source code. We categorized 28 of 122 members as critical and chose those for protection and partitioning. We divided `task_struct` into two parts: `task_struct` containing security sensitive members, and `insecure_task_struct` containing all non-sensitive fields.

Before partitioning, memory was allocated to an instance of the `task_struct` via `kmem_cache_alloc`. This function returns a pointer to an object of type `task_struct` from the slab cache [10]. The retrieved memory block might cross page boundaries, consequently making it difficult to provide protection for only those members that need it. Hence, we changed the memory allocation to instead use `get_free_pages` and `kmalloc`. Using `get_free_pages`, we allocated each `task_struct` on a complete page, thereby separating the critical members from non-critical fields in the kernel memory; we allocated each instance of `insecure_task_struct` using `kmalloc`. As described previously, we connected the two substructures by maintaining a reference from `task_struct` to `insecure_task_struct`. Finally, we modified Linux's `free_task` function to deallocate the memory pages allocated to `task_struct` and `insecure_task_struct` separately. After the structure partitioning, we modified the Linux kernel source code in order to work with the partitioned structure. Our source code modification to the Linux 2.6.16 kernel altered 2536 SLOC, which is 0.036% of the total number of lines of code in the kernel (7,041,452 SLOC).

6.5.1.2 Partitioning of `module`

We partitioned Linux's `module` structure using our structure alignment technique. We categorized 2 of the structure's 29 members as critical and separated them from the non-critical members with an alignment buffer that places the critical fields on a different page than non-critical fields at runtime. We first grouped all the security-critical members in the module data structure together by reorganizing the data structure. Our new alignment field provided padding that filled the rest of the memory

page and caused the non-critical members to cross the page boundary to a new, unprotected page. This alternative partitioning technique did not require creation of a new insecure structure as done for the `task_struct`. The approach required no source code modification beyond the addition of the alignment buffer and required only a straightforward recompilation of the kernel due to the changed field offsets within the `module` structure.

6.5.2 Access Mediation and Policy Enforcement

The memory protection software operates in two phases, both occurring concurrently: a management phase when the kernel adds or removes a page frame number (PFN) to or from the list of protected PFNs, and the mediation phase providing the actual memory protection. To perform addition, removal, and lookup operations on a PFN, we created new APIs inside the Xen hypervisor. The API includes the functions `addPFNToDB`, `removePFNFromDB`, and `checkPFNInDB`, providing us with the ability to add, remove, and find a protected PFN, respectively.

The second phase works by modifying the shadow page table (SPT) code of Xen. The SPT is the native page table used by the memory management hardware and managed by Xen. To provide memory protection, we modify the `__sh_propagate` function. When a new memory page is added to the guest page table, `__sh_propagate` propagates this entry to the SPT to keep both the tables in sync. While propagating this update, the memory protection system intercepts the update and checks whether the propagation involves a protected page. If a page belongs to the list of protected pages, it removes the write permission from the page by setting the page write bit to zero in the SPT.

To intercept subsequent write faults on a protected page, we modified the shadow page fault handler function called `sh_page_fault`. When a fault occurs, our code inside `sh_page_fault` verifies whether or not the fault is a result of its protection

mechanism. The verification dictates how the fault is to be processed by our software. If the fault is due to some other activity in the guest, then we ignore it and resumes the guest's normal operation. Otherwise, we look into the fault to determine what code region is attempting to write to the protected page, as described in Section 6.4.3.

6.5.3 Instruction Emulation

When the kernel memory access control policy permits a mediated write operation, we must reproduce the effects of the operation in a guest operating system's memory. This functionality resides in our software because the guest operating system cannot execute the write operation itself due to the write protection bit set on the faulted page. We implemented an instruction emulator inside Xen to perform the emulation of memory writes. With this emulator, we can replay attempted writes when subsequently determined legitimate. To emulate an instruction, our software needs to first locate the instruction and then fetch it from guest memory. To achieve this, we use Xen's function called `hvm_copy_from_guest_virt` that reads and writes to arbitrary guest locations.

When a faulting instruction is fetched, the emulator decodes and executes the instruction inside Xen. Depending upon the instruction type, the decoding process may identify source and destination operands. The emulator executes the instruction by reading and writing the memory locations directly from the hypervisor. To ensure transparency to the guest operating system, it updates all context registers including the instruction pointer to point to the next instruction.

6.5.4 Execution History Extraction

The execution history extractor of our software performs a walk on the kernel stack of the guest operating system by mapping the guest kernel's stack pages into the hypervisor's memory and then traversing stack frames present on the pages. Performing a walk on the kernel stack is challenging due to the unstructured layout of call stacks

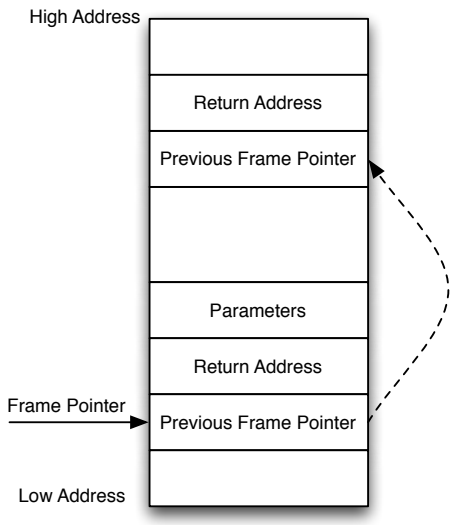


Figure 33: Guest Kernel Stack Walk from the Xen Hypervisor.

on x86 systems and the presence of interrupt stack frames. To successfully extract stack frames, we compiled our guest kernel with a compile-time option that produces kernel code maintaining a stack frame base pointer (`ebp` register) throughout the kernel's execution. Fortunately, interrupt stack frames do not pose problems for us because we only perform stackwalks following a page fault that occurred due to our protection; we do not perform a stackwalk at arbitrary points of the kernel's execution. Our software pauses the guest kernel while executing the stackwalk to ensure that no modifications occur while reading the guest's memory state.

When a page fault occurs, the extractor finds the location of the current stack frame from the `ebp` register. It subsequently determines the location of the return address by adding 4 bytes to the value of `ebp`. Figure 33 shows the layout of the kernel stack. To get the actual return address, the extractor reads the value present on the computed return address location. To get the previous frame, it extracts the address stored at the location pointed to by the `ebp` register's contents. The extractor repeats this process until it reaches the end of the stackwalk.

Table 20: Our attack detection results against Linux-based rootkits that modify the kernel's process and module data structures. A $\surd$ indicates that the rootkit exhibited a particular malicious behavior.

Name	*Hidden Process*	*Hidden Module*	*Privilege Escalation*	*Result*
hp	$\surd$			Detected
all-root			$\surd$	Detected
kbd-version2			$\surd$	Detected
kbd-version3			$\surd$	Detected
override			$\surd$	Detected
synapsys			$\surd$	Detected
rkit			$\surd$	Detected
lvtes		$\surd$	$\surd$	Detected
adore-ng		$\surd$	$\surd$	Detected

6.6 Evaluation

In this section, we evaluate both the attack detection capability and the performance of our kernel data protection software. We also perform detailed false positive and security analysis of our protection software.

6.6.1 Attack Prevention and Detection

Our software prevents unauthorized modification of security-critical kernel data structures. Note that we are able to protect both static and dynamic kernel data structures. However, our experiments focus on rootkit identification based upon their attempted alteration of dynamic kernel data, as this represents a significant new advancement in defensive capabilities.

We tested our software against a collection of DKOM rootkits present in the wild. Table 20 shows our Linux rootkit samples and the malicious behaviors that they exhibit. During our testing, we ran each rootkit sample in the user VM, which was running our Linux 2.6.16 kernel with partitioned `task_struct` and `module` data structures. Our protections started at kernel boot so that all processes beginning with the `init` process and all modules can be protected. Below, we provide a detailed

description of detection of the `lvtes` keylogger and the `all-root` rootkit.

Lvtes hides a malicious module by removing it from the doubly-linked module list. It uses the kernel macro `list_del`, which directly deletes a module from the list by modifying a member called `list` in the `module` structure. This member stores next and previous pointers of the list. To test our protection against `lvtes`, we inserted the keylogger module in the user VM's kernel. When the rootkit tried manipulating `list.next`, and `list.prev` to remove the module from the doubly-linked list, it caused a page fault because `list` is considered to be a sensitive member and is protected by our software. Our enforcement mechanism verified whether the access should be allowed or denied by checking which code attempted to modify protected members. We looked at the instruction pointer, which in this case belonged to an untrusted code region in a module. Consequently, we denied the access to the rootkit.

The `all-root` rootkit [107] directly modifies the uid, gid, euid, and egid members of a `task_struct` structure. These members determine the privilege level of a process, which in turn restricts the types of execution that a process can perform. When the rootkit loads, it hooks into the system-call table and replaces the function pointer associated with the `getuid` system call with its malicious function `mal_getuid`. When a malicious process cooperating with the rootkit executes `getuid`, the rootkit's `mal_getuid` function will execute instead and will set the uid, gid, euid, and egid of the process to 0. This escalates the privileges of the invoking process to a superuser. We tested the protection software with this attack and detected this modification because these fields were protected. Modifications to these fields caused a page fault, and the data protection software found that the modifying code was in a module and denied the access. Our results, shown in Table 20, indicate that our kernel data protection software provided a 100% detection rate for DKOM rootkits.

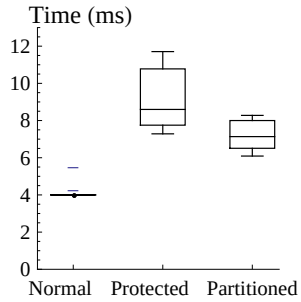


Figure 34: Module loading operation performed via `insmod`; Here, "Normal" refers to measurements that have no kernel memory protection, "Protected" includes memory protection without partitioning, and "Partitioned" includes both memory protection and partitioning. Smaller measurements are better.

6.6.2 Performance

We carried out several experiments to measure the performance overhead incurred by the protection offered by our software as well as to show the effect of partitioning. Our results were measured on an Intel Core 2 Quad 2.66 GHz system with Fedora 8 in the security VM and our partitioned 32-bit Linux 2.6.16 kernel in the user VM. We assigned 1GB of memory to the user VM and 3GB total to the security VM and Xen. Unless otherwise specified, we performed the experiments reported in tables five times and present the median value. We show the results of some experiments using boxplots due to measurement variations common to virtualized environments. In all the experiments, "normal" refers to measurements that have no kernel memory protection, "protected" includes memory protection without partitioning, and "kernel data protection" includes both memory protection and partitioning.

We first evaluated the effect of protection and partitioning of module structures on Linux modules by measuring the time taken by module load (`insmod`) and unload (`rmmod`) operations. We wrote a sample module that traverses the list of loaded modules and inserted it using the `insmod` program. The same module is then unloaded using the `rmmod` program. Figures 34 and 35 present the results, and it can be seen that the loading and unloading time is higher for the unpartitioned kernel when compared with our system's time.

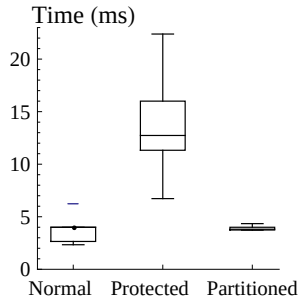
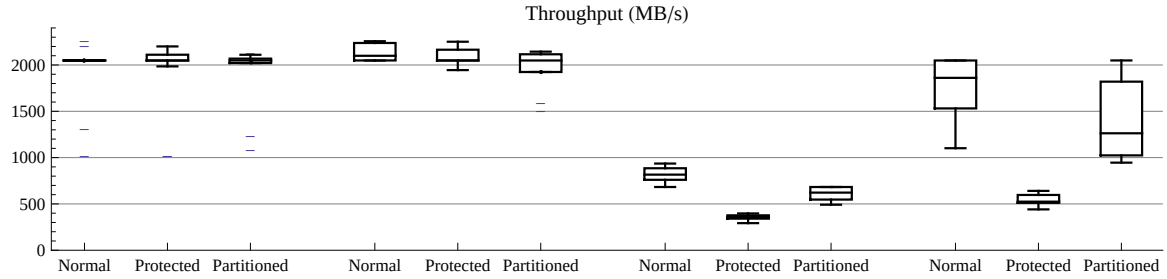


Figure 35: Module unloading operation performed via `rmmod`; Here, "Normal" refers to measurements that have no kernel memory protection, "Protected" includes memory protection without partitioning, and "Partitioned" includes both memory protection and partitioning. Smaller measurements are better.



(a) Read (b) Re-read (c) Write (d) Re-write

Figure 36: Performance impact of kernel memory protection on use of the kernel's file cache. All measurements show throughput in MB/s; higher measurements are better. Here, "Normal" refers to measurements that have no kernel memory protection, "Protected" includes memory protection without partitioning, and "Partitioned" includes both memory protection and partitioning. Boxes show medians and first and third quartiles. Outliers appear as dashes. Groupings show performance of (a) file-cache cold reads, (b) cache-warm reads, (c) cache-cold writes, and (d) cache-warm writes.

In the next set of experiments, we measured the effect of our data protection on filesystem cache performance. To test this, we used a filesystem benchmark called `IOzone` [161], which measures the throughput of read, write, re-read, and re-write operations. Figure 36 shows quartile measurements of ten repetitions of each test. Note that these results show use of the kernel's filesystem cache—rather than of disk operations—due to high variability of disk performance measurements in a virtualized environment. Our protection incurs less overhead on file cache operations when performed using the partitioned kernel as compared to its effect on unpartitioned kernel, and in many cases, its performance nears that of the unprotected system.

Table 21: Performance impact of our protection system on real-world applications; smaller measurements are better.

Operations	*Normal VM (sec)*	*Protected (sec)*	***Overhead*** *%*	*Kernel Data Protection (sec)*	***Overhead*** *%*
Compression	41.91	43.41	3.58	42.03	0.29
Decompression	10.09	11.68	15.76	10.11	0.20
Network File Transfer	18.46	19.45	3.85	18.52	0.33
Kernel Compilation	56.39	65.47	16.10	58.45	3.65
Apache Compilation	5.88	7.42	26.19	6.06	3.06

Next, we measured the memory usage of both the partitioned and unpartitioned kernels. We performed this experiment by iterating through the guest kernel's page tables. At any given time, the partitioned kernel used 6502 pages as compared to 6302 pages used by the unpartitioned kernel. Though this extra memory is less than 1MB, it can be further reduced by using an improved memory allocator.

We next measured our data protection software's performance on real world software by world software by testing it with full applications. In our experiments, we performed compilation of a stripped down version of the Linux kernel, compilation of the Apache webserver, compression and decompression of a 225 MB file, and a network copy operation using wget. Table 21 shows the results of experiments, and it is evident that our partitioning has significantly reduced the overhead caused by its protection when compared with the overhead on an unpartitioned kernel.

Finally, we measured the effect of protection and partitioning of a process data structure. We measured the effect with two tests that exercise heavy legitimate use of the protected structures—process creation and context switch time—using the lmbench [78] Linux benchmark tool. Lmbench performs three different experiments to measure the cost of process creation. Our results, shown in Table 22, indicate

Table 22: Process creation and context-switch time measured with lmbench; smaller measurements are better. The default CPU time-slice on the test system was 100 ms.

Operations	*Normal VM (μs)*	*Protected (μs)*	***Overhead (%)***	*Kernel Data Protection (μs)*	***Overhead (%)***
Process fork+exit	342.89	1255.89	266.27	550.52	60.55
Process fork +execve	355.69	1296.08	264.38	586.13	64.78
Process fork +/bin/sh -c	2482.30	6217.77	150.48	2946.39	18.69
Context switch	1.85	41.14	2123.78	6.45	248.65

that we incur low overhead on a partitioned kernel as compared to the overhead on an unpartitioned kernel. Though this overhead of our software is more in percentage when compared with the "Normal VM" time, it gets amortized across the lifetime or execution of a process.

Although we have not yet explored many optimizations, the above results provide strong evidence that our protection software can provide a balance between security and performance. Our software's performance on several applications is encouraging and suggests that this approach is practical.

6.6.3 Potential Performance Improvements

Although the previous section shows the overhead of the data protection on real world applications to be low, its performance can further be improved by providing efficient ways to do common operations. We have identified the following improvement opportunities:

6.6.3.1 New Memory Allocator

When using structure division partitioning, we allocate a security-critical structure on a protected page using the kernel function `get_free_page`, which returns a new page from the kernel memory pool. Since the security-critical portion of each data

structure is allocated on a new page, this allocation wastes system memory because these pages are not used further.

An improved memory allocator can better utilize a protected page by allocating more secure structures on the same page until the page gets filled. This new allocator can be a wrapper around the existing kernel memory allocator and can reuse an already allocated memory page to store new security-critical objects. When one page gets filled, the kernel allocates a new page for critical structures. This approach provides two advantages: first, it reduces the TLB flushing that happens inside Xen when we add a new page to the list of protected pages. Second, this approach reduces the number of VMCALLs from the guest to Xen requesting protection on a new page because the page where a new structure will be located may already be protected. The downside of reusing a page is that it leads to increased page faults during the initialization of a new structure. Another improvement is along the line of object caching performed by commodity operating systems kernel. This object caching can also be applied to critical data structures by creating memory slabs for them [10].

6.6.3.2 Mapping Memory Pages

We use the memory mapping and unmapping functions of Xen, such as `hvm_copy_to_guest_virt` and `hvm_copy_from_guest_virt` during the process of stackwalk and instruction emulation. These functions access the guest VM's memory from the hypervisor, however, Xen provides a very inefficient implementation. On each invocation of these functions, Xen maps a requested page from the guest kernel's memory into the hypervisor memory, performs the operation, and then unmaps the page.

An improvement can be made in the implementation of these functions by keeping a memory page mapped inside the hypervisor memory to provide locality of reference. For example, during a stackwalk, we map and unmap a stack page everytime when

it accesses a return address and frame pointer. If the page were mapped once during a single walk, then Xen would have avoided multiple mapping and unmapping operations.

6.6.4 False Positive Analysis

A false positive in our approach occurs when a security-critical member is modified by a benign module or driver that violates our integrity policies described in Section 6.4.1. Our analysis revealed the following:

- There were no instances when security-critical kernel data protected by our software was *directly* modified by a benign driver.
- Whenever security-critical data protected by the data protection software was altered by a benign driver, it was done using *trusted upgraders* designed by kernel developers for that purpose, and they left the kernel in a consistent state. We illustrate this point with an example: a `task_struct` contains a member called `run_list`, which is similar to `tasks` (pointer to accounting list), but contains next and previous pointers for job scheduling; we protect the `run_list` member. These pointers are modified by functions such as `enqueue_task` and `dequeue_task`, which in turn are called from the `schedule` function, which is a trusted upgrader. The `schedule` function is invoked on each context-switch and modifies the run list; it is also invoked from all modules. Since our policy allows changes made to kernel data via trusted upgraders, whenever members such as `run_list` were modified, we verified the call-chain and allowed the modification.

With the above design in place, our software did not show any false positives and detected all attacks.

6.7 Discussion

In this section, we discuss some extensions to our protection software that address possible attacks and spread the protection to more data structures and to different operating systems.

6.7.1 Possible Attacks

An attacker cannot bypass our protection software because any write operation on a protected page invokes the data protection system. However, an attacker may manipulate the kernel stack to force our software to conclude that the request has originated from trusted code regions and alter indirect jump targets to regain execution control. Stack manipulation attacks can be thwarted by maintaining a shadow stack, a parallel stack that resides in a protected region and stores correct return addresses and frame pointers [12, 25, 41]. Attacks involving indirect jumps can be detected by kernel control flow integrity [115].

6.7.2 Protection of other kernel data structures

Using our prototype implementation, we showed how to protect two important kernel data structures. We chose these data structures because of their size, pervasiveness, and importance. However, there are other kernel data structures that may also require similar protection. Our software can be extended to protect other data structures. Our performance results show that the data protection system imposes very low overhead on the two protected structures, and we expect that we will maintain its low overhead even when it extends its protection to other structures. However, the actual performance overhead depends upon the number of sensitive members, and the rate at which the kernel and its components legitimately modify these sensitive members in newly-protected structures.

6.7.3 Windows operating system support

Our current implementation requires the source code of an operating system in order to partition a structure into secure and insecure parts. This kind of protection is difficult to design for a closed source operating system such as Windows. However, the solution presented in this paper to protect kernel dynamic data can be adopted by Windows developers by creating a partitioned kernel during an OS development cycle to support such protection mechanisms. Although this approach requires code revisions, they are a one-time design cost borne by the kernel developers that provide long-term improvements to a kernel's security.

6.8 Conclusions

We developed our protection software to provide partitioned kernel memory in a manner similar to memory isolation provided by the kernel for its applications. We protect security-critical data by protecting memory pages containing that data. To provide balance between security and performance, we altered the kernel memory layout to aggregate data needing the same policy enforcement on the same memory pages. Our security evaluation shows that the system is capable of detecting attacks against dynamic kernel data, and its performance evaluation shows low overheads on microbenchmarks and real-world applications.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Summary

Computer security issues have always been around, and they will continue to exist in future. To combat against these attacks, users deploy anti-virus software in their machines, and network administrators run intrusion detection systems at the perimeter of a network. The current malware detection approaches either operate from networks or within hosts. Host-level security solutions are susceptible to direct attacks, evasion from kernel-level malware, and false positives due to parasitic malware. On the other hand, network-level solutions are unable to provide fine-grained attribution of malicious behaviors to the actual malicious code. My dissertation research investigated a new direction that combines information present at multiple places. We correlate information visible at networks with the host-level software execution information to be able to perform better malware detection. Solving the malicious code identification problem required solutions to additional challenging sub-problems.

First, we provided a new tamper-resistant process attribution software that attributes malicious network traffic to host-level processes bound to the traffic. Our software operated from trusted virtual machines to remain protected from direct attacks, and bridged the semantic gap by using virtual machine introspection. We made the following contributions in designing the process attribution software:

- We performed correlation between network flows and processes from outside the monitored virtual machine to identify malicious code running inside the monitored VM.

- We designed and implemented a tamper-resistant software and evaluated its

effectiveness with benign and malicious programs. The performance results showed that our software incurs low overhead.

- We extended the design of the process attribution software to create a new tamper-resistant, application-aware firewall for virtualized environments.

The problem of identifying the end-point of a connection inside hosts gets complicated due to the presence of parasitic malware. Parasitic malware obscures their presence by injecting malicious code into benign processes at runtime. This kind of malicious modifications also evade security utilities that attempt to verify the integrity of program images at disks. We thoroughly explored various parasitic attacks and categorized them into user- and kernel-level behaviors. To detect a user-level process-to-process parasitic attack, we model it as sequences of Windows API calls and detect the attacks by monitoring non-bypassable system-calls invoked by processes. However, detecting kernel-level parasitic is challenging due to the absence of any driver monitoring interface inside the kernel. We created a new security monitor that records all kernel APIs invoked by drivers and detects kernel-level parasitic attacks. We made the following contributions for detecting user- and kernel-level parasitic attacks:

- We correctly attributed observed behaviors to the actual malware responsible for creating those behaviors, even in the presence of parasitic malware that injects code into benign applications.

- To monitor parasitic behaviors at the user-level, we designed and implemented software that monitors system calls from the hypervisor.

- To monitor kernel-level parasitic attacks, we created a non-bypassable driver monitoring interface inside the kernel. We imposed the non-bypassable kernel interface upon dynamically-loaded device drivers thereby preventing control flows from drivers into arbitrary kernel code.

- We efficiently handled control flows spanning the kernel interface barrier via on-demand dynamic binary rewriting and runtime code generation. These actions reduced world switches into and out of the hypervisor without compromising the security of the system.

- We evaluated our software by demonstrating its ability to detect kernel-level and user-level parasitic attacks. The performance evaluation demonstrated that even with runtime on-host monitoring, our performance impact remains low.

Finally, we presented the design and techniques of the data protection software that protects the integrity of kernel data from malicious modifications. This is important due to reliance of the process attribution software on the kernel data to correctly attribute the malicious traffic to host-level processes. We made the following contributions:

- We created protected memory regions within the unified kernel data space. A kernel could then isolate its security-critical data from kernel components having low trust, creating assurance in the critical state.

- We showed how to optimize kernel memory space layout for the protection constraints created by our software. Our layout changes did not impact correctness of kernel execution, but they allowed our access control enforcement to operate with higher performance.

7.2 Open Problems

This dissertation has addressed many research problems related to malicious code detection, and it has also created new research problems and directions that are worth pursuing to improve the effectiveness of our defenses.

7.2.1 Host and Network Events' Correlation

We demonstrated the usefulness of correlating network packets to on-host processes by identifying the malicious code responsible for the traffic. In our architecture, we rely on network-level security software to detect attacks and trigger our host-level software to identify malicious process bound to the malicious traffic. In this design, if a NIDS exhibits false positives, our host-level component will still report the process bound to a connection as malicious. Future improvements in the current architecture may include a probabilistic model that assigns suspicion scores to network- and host-level behaviors exhibited by processes. With this improved design, network-level security software would assign a score to each traffic instead of classifying it as an attack and wait for the host-level software to boost the score based on host-level behaviors to confirm the attack.

In another follow up work, it would be interesting to explore other scenarios where the host and network correlation information could potentially be useful. The current network intrusion detection systems deployed in enterprises are overloaded with traffic that they have to process to detect attacks. The host-level information could reduce this burden by providing knowledge about potentially malicious traffic based on the information associated with processes and their provenance. For example, the host-level agent can extract information about a process, such as who installed this process: a user or browser, when did it get installed, what changes did it make on the system after that, and other execution related information, and pass it to the NIDS. Using this information, the NIDS can prioritize the traffic they want to look into immediately for attacks and delay the processing of other traffic.

Similar solutions can be adopted for other environments such as mobile phones. Today, mobile phones are not only used for telephony operations, but they are considered as general-purpose computing platforms. These phones run complex operating systems and frequently interact with the backend servers. Due to the ability to store

private confidential data and perform critical operations such as banking, attackers started targeting these devices for monetary gains. To identify the on-phone malicious code, we can employ our correlation approach in these environments too. It would be challenging to convince mobile phone users to deploy an agent on their phones, but an idea of providing our agent as an application through the app store seems plausible.

7.2.2 Fine-grained Remediation

The coarse-grained information provided by the network-level software alone permits only coarse-grained responses: an administrator could excise an infected system from the network, possibly for re-imaging. Unfortunately, in many common use scenarios, complete disk sanitization results in intolerable losses of critical data not stored elsewhere, even though that data may not have been affected by the infection. On the other hand, the attribution of malicious network behaviors to host-level processes creates the foundation for fine-grained remediation. By knowing the malicious software, administrators can initiate the targeted responses in which they only restart the victim process and delete parasitic malware instances. In this thesis, we have not explored the automatic remediation procedures. Future research projects can consume the information provided by our software to develop methods to remove the malicious code automatically. As described in Chapter 4, our software also identifies objects such as files and handles utilized by malicious processes. With this knowledge, remediation procedures can also delete resources used by malicious processes to offer fine-grained remediation. The challenge lies in performing this task without loosing users' private data or deleting important system files.

7.2.3 Kernel Data Protection using Address-Space Partitioning

Our kernel data protection software, presented in Chapter 6, relies on the kernel stack to extract the execution history information, and hence it is susceptible to attacks

that modify information present on it. An alternative method of extracting the execution history is to monitor drivers interaction with the kernel code. As described in Chapter 5, our kernel API monitor isolates drivers in an address space separate from the core kernel and monitors the kernel code invocation by drivers through the exported interface. With this design, whenever a driver enters the core kernel through the exported interface, that transition will be monitored by our software. This information can be used by the data protection software to determine the provenance of a request to update the critical kernel data.

In this new design, we can map kernel data pages, containing security critical data as read-only in the drivers' address space. These pages can be read and written in the kernel's address space. With these permissions, when a driver code attempts to directly modify any critical data, there will be a fault due to write protections. The kernel data protection software inside the hypervisor receives the control of execution and disallows the write operation. Also, any attempt to modify the data using exported functions will be verified before allowing it to go through the non-bypassable interface. This driver isolation based design will defeat the stack modification attacks.

7.2.4 Kernel Malware Analysis

Current malware analysis research has developed various approaches for analyzing user-space malware. However, there is a limited research on the analysis of kernel-space malware. Recent years have seen a surge in kernel malware instances, hence it is important to analyze kernel malware effectively. Analyzing kernel malware is challenging due to a unified address space inside the kernel; malware instances can invoke the kernel code and modify the kernel data at their will. This dissertation has presented software for the detection of kernel-level parasitic attacks and the protection of kernel data. We can use our kernel monitoring software to contain a malicious kernel driver in a separate address space. With this isolation, when the malicious

driver executes the kernel code, reads, or updates the kernel data, such operations will be monitored by our software. Based on this log, we can generate access profile for the malware, create its signature, and use this signature to detect it in the future.

7.2.5 Improving Kernel Reliability

We describe mechanisms to interpose on transitions from drivers into the core kernel and vice versa by isolating drivers in a separate address space. Further, we described how can we reduce the overhead by utilizing on-demand code generation and binary rewriting of kernel and driver code. The ability to perform code generation and binary writing in the kernel at runtime offers an excellent opportunity to develop tools to improve the reliability of systems. As per studies, drivers are the source of most systems' crashes and bugs [20]. Despite these problems, not many tools and utilities are available that can test drivers for the presence of bugs and vulnerabilities.

An extension to our infrastructure can be used to create fault injection utilities for drivers in the same way as we have utilities for user-space applications and libraries [90]. This system will interpose on drivers' execution and modify parameters passed to drivers' functions at runtime. For example, we can modify parameters present on the stack by changing a pointer value to null before a function is invoked. With this null value if a driver crashes, it means that the null value check was not present in the driver, and the code must be fixed. Apart from fault injections, our software can also be extended to generate other execution statistics inside the kernel such as optimizing frequently invoked functions to remove bottlenecks. However, challenges remain to make these tools work in the kernel in the presence of interrupts and other asynchronous operations.

7.2.6 Cloud Security

My dissertation research solves client-side systems security issues using virtualization technologies. Due to the extensive use of virtualization to support cloud computing, it would be interesting to investigate security issues in cloud environments. In a self-hosted computing environment, customers or organizations own all the resources, including the code, data, and computation. As customers move to third-party managed public cloud environments, they relinquish control over their assets. Public clouds provide services to a variety of customers and organizations. There are numerous ways in which the confidentiality and integrity of customers resources in this potentially untrusted environment can be compromised. This includes operational errors and unintentional mistakes due to large trusted computing base, and insider attacks from hostile administrators [52]. Lack of any basis of verifiable trust between the cloud providers and the customers leads to security being a main concern in cloud computing [34].

An interesting research direction would be to pursue projects that attempt to bridge the trust gap between the providers and customers. We have to build systems that allow customers to control their resources in the cloud. They should be able to provide arbitrary security policies on their resources, and providers must enforce those policies. Another research direction should be on creating infrastructure with which providers can verifiably inform their customers about how their data is processed, stored, and migrated in the cloud. For example, how would a provider assure customers that their data is backed up three times on three different disks, not three times on the same disk. Bridging the trust gap is a challenging problem, and its solution would definitely help in the wider adoption of cloud computing.

Apart from insider threats, cloud environments also suffer from attacks caused by malware. Attackers are using cloud VMs for hosting malware, sending spam, and launching denial-of-service attacks [37]. To protect the customers' data and cloud

infrastructure, it is also important to identify the malicious code running inside cloud VMs. One potential research direction is to investigate the challenges of using our system in cloud environments. The challenges posed by cloud environments, such as third party managed environments and migration of customer VMs at runtime, make the problem interesting and worth exploring.

7.3 Closing Remarks

This thesis addressed important research problems and presented practical solutions for them. In particular, we demonstrated the benefits of identifying malicious code by combining host- and network-level information. We designed a practical architecture that utilizes virtualization technologies to detect the malicious code in a robust and secure manner in the presence of user- and kernel-level parasitic malware. Our architecture relies on memory virtualization features supported by all recent hypervisors such as VMware [155], KVM [77], and Xen [8]. Further, due to the recent research on the design of thin, security-purpose, and client-side hypervisors [130, 133] that also support memory virtualization, our architecture may find its place on end users' desktops in future. Together, our contributions produce a unified research outcome – an improved malicious code identification system for user- and kernel-level malware. Due to the dynamic nature of security area, it is important to understand and adapt to new environments and threats. We believe that the approaches presented and open problems discussed in this thesis will definitely be helpful in guiding the future research in this space.

REFERENCES

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A new kernel foundation for unix development," in *USENIX Conference*, (Atlanta, GA), 1986.

[2] Adams, K. and Agesen, O., "A comparison of software and hardware techniques for x86 virtualization," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (San Jose, CA), Oct. 2006.

[3] Alexander Tereshkin, "Rootkits: Attacking personal firewalls." `www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf`. Last accessed May 15, 2011.

[4] Ames, S. A., Gasser, M., and Schell, R. R., "Security kernel design and implementation: An introduction," *IEEE Computer*, vol. 16, no. 7, 1983.

[5] Anderson, J. P., "Computer security technology planning study," in *Technical Report ESD-TR-73-51*, (The MITRE Corporation, Badford, MA), 1972.

[6] Axelsson, S., "The base-rate fallacy and its implications for the difficulty of intrusion detection," in *ACM Conference on Computer and Communications Security (CCS)*, (Singapore), Nov. 1999.

[7] Baliga, A., Ganapathy, V., and Iftode, L., "Automatic inference and enforcement of kernel data structures invariants," in *Annual Computer Security Applications Conference (ACSAC)*, (Anaheim, CA), Dec. 2008.

[8] 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 *ACM Symposium on Operating System Principles (SOSP)*, (Bolton Landing, NY), Oct. 2003.

[9] Biba, K. J., "Integrity considerations for secure computer systems," Tech. Rep. MTR-3153, Mitre, Apr. 1977.

[10] Bonwick, J., "The slab allocator: An object-caching kernel memory allocator," in *USENIX Summer Technical Conference*, (Boston, MA), June 1994.

[11] Borders, K., Zhao, X., and Prakash, A., "Siren: Catching evasive malware," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2005.

[12] Broadwell, P., Harren, M., and Sastry, N., "Scrash: A system for generating secure crash information," in *USENIX Security Symposium*, (Washington, D.C), Aug. 2003.

[13] Brumley, D. and Song, D., "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, (San Diego, California), Aug. 2004.

[14] Castro, M., Costa, M., Martin, J., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R., "Fast byte-granularity software fault isolation," in *ACM Symposium on Operating System Principles (SOSP)*, (Big Sky, Montana), Oct. 2009.

[15] CERT, "Cert statistics." `http://www.cert.org/stats/`. Last accessed May 25, 2011.

[16] Check Point, "ZoneAlarm." `http://www.zonealarm.com/store/content/home.jsp`. Last accessed May 15, 2011.

[17] Chen, P. M. and Noble, B. D., "When virtual is better than real," in *Hot Topics in Operating Systems (HotOS)*, May 2001.

[18] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R., "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Seattle, WA), Mar. 2008.

[19] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X., "Secure web applications via automatic partitioning," in *ACM Symposium on Operating System Principles (SOSP)*, (Stevenson, WA), Oct. 2007.

[20] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D., "An empirical study of operating system errors," in *ACM Symposium on Operating Systems Principles (SOSP)*, (Banff, Canada), Oct. 2001.

[21] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E., "Semantics-aware malware detection," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2005.

[22] Community Developers, "Ebtables." `http://ebtables.sourceforge.net/`. Last accessed May 15, 2011.

[23] Community Developers, "Tripwire." `http://sourceforge.net/projects/tripwire/`. Last accessed May 15, 2011.

[24] Corbato, F. and Vyssotsky, V., "Introduction and overview of the Multics system," in *Fall Joint Computer Conference*, (Las Vegas, NV), Nov. 1965.

[25] Corliss, M. L., Lewis, E. C., and Roth, A., "Using DISE to protect return addresses from attack," in *Workshop on Architectural Support for Security and Anti-Virus*, (Boston, MA), Oct. 2004.

[26] Criswell, J., Geoffray, N., and Adve, V., "Memory safety for low-level software/hardware interactions," in *USENIX Security Symposium*, (Montreal, Canada), Aug 2009.

[27] Cyveillance, "Cyveillance testing finds av vendors detect on average less than 19% of malware attacks." `http://www.cyveillance.com/web/news/press_rel/2010/2010-08-04.asp`. Last accessed May 15, 2011.

[28] Debar, H. and Wespi, A., "Aggregration and correlation of intrusion-detection alerts," in *Recent Advances in Intrusion Detection (RAID)*, (Davis, CA), Oct. 2001.

[29] Dinaburg, A., Royal, P., Sharif, M., and Lee, W., "Ether: Malware analysis via hardware virtualization extensions," in *ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Oct. 2008.

[30] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W., "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2011.

[31] Economics, C., "Annual worldwide economic damages from malware exceed 13 billion dollars." `http://www.computereconomics.com/article.cfm?id=1225`. Last accessed May 15, 2011.

[32] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C., "XFI: Software guards for system address spaces," in *Operating Systems Design and Implementation (OSDI)*, (Seattle, WA), Nov. 2006.

[33] Erlingsson, Ú. and Schneider, F. B., "Irm enforcement of Java stack inspection," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2000.

[34] EweekEurope, "Security main concern around cloud planning." `http://www.eweekeurope.co.uk/news/news-security/security-main-concern-around-cloud-planning-1841`. Last accessed May 15, 2011.

[35] Feng, H., Kolesnikov, O., Fogla, P., Lee, W., and Gong, W., "Anomaly detection using call stack information," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2003.

[36] Feng, H. H., Kolesnikov, O. M., Fogla, P., Lee, W., and Gong, W., "Anomaly detection using call stack information," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2003.

[37] Fisher, D., "Attackers using amazon cloud to host malware." http://threatpost.com/en_us/blogs/attackers-using-amazon-cloud-host-malware-060611. Last accessed May 15, 2011.

[38] Ford, B. and Cox, R., "Vx32: Lightweight user-level sandboxing on the x86," in *USENIX Annual Technical Conference (ATC)*, (Boston, Massachusetts), June 2008.

[39] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A., "A sense of self for UNIX processes," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, California), May 1996.

[40] Fraim, L., "SCOMP: A solution to the multi-level security problem," *IEEE Computer*, vol. 16, no. 7, 1983.

[41] Frantzen, M. and Shuey, M., "StackGhost: Hardware facilitated stack protection," in *USENIX Security Symposium*, (Washington, DC), Aug. 2001.

[42] Ganapathy, V., Jaeger, T., and Jha, S., "Automatic placement of authorization hooks in the linux security modules framework," in *ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Mar. 2005.

[43] Ganapathy, V., Jaeger, T., and Jha, S., "Retrofitting legacy code for authorization policy enforcement," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), Mar. 2006.

[44] Ganapathy, V., Renzelmann, M. J., Balakrishnan, A., Swift, M. M., and Jha, S., "The design and implementation of microdrivers," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Seattle, WA), Mar. 2008.

[45] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D., "Terra: A virtual machine-based platform for trusted computing," in *ACM Symposium on Operating Systems Principles (SOSP)*, (Bolton Landing, NY), Oct. 2003.

[46] Garfinkel, T. and Rosenblum, M., "A virtual machine introspection based architecture for intrusion detection," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2003.

[47] Garfinkel, T., Rosenblum, M., and Boneh, D., "Flexible OS support and applications for trusted computing," in *Hot Topics in Operating Systems (HOTOS)*, (Lihue, Hawaii), May 2003.

[48] Gerald Oskoboiny, "Whitelist-based spam filtering." http://impressive.net/people/gerald/2000/12/spam-filtering.html. Last accessed May 15, 2011.

[49] Giffin, J., Jha, S., and Miller, B., "Detecting manipulated remote call streams," in *USENIX Security Symposium*, (San Francisco, California), Aug. 2002.

[50] Giffin, J. T., Jha, S., and Miller, B. P., "Efficient context-sensitive intrusion detection," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2004.

[51] Giffin, J. T., Jha, S., and Miller, B. P., "Automated discovery of mimicry attacks," in *Recent Advances in Intrusion Detection (RAID)*, (Hamburg, Germany), Sept. 2006.

[52] Godschalk, Z., "Cloud security alliance." `http://blog.cloudsecurityalliance.org/2011/03/21/three-cloud-computing-data-security-risks-that-can%E2%80%99t-be-overlooked/`. Last accessed May 15, 2011.

[53] Gold, B. D., Linde, R. R., and Cudney, P. F., "Kvm/370 in retrospect," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 1984.

[54] Gopalakrishna, R., Spafford, E. H., and Vitek, J., "Efficient intrusion detection using automaton inlining," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2005.

[55] Grok, "Coromputer Dunno." `http://lists.grok.org.uk/pipermail/full-disclosure/attachments/20070911/87396911/attachment-0001.txt`. Last accessed May 15, 2011.

[56] Gu, G., Porras, P., Yegneswaran, V., Fong, M., and Lee, W., "BotHunter: Detecting malware infection through IDS-driven dialog correlation," in *USENIX Security Symposium*, (Boston, MA), Aug. 2007.

[57] Gu, G., Zhang, J., and Lee, W., "BotSniffer: Detecting botnet command and control channels in network traffic," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2008.

[58] Gummadi, R., Balakrishnan, H., Maniatis, P., and Ratnasamy, S., "Not-a-bot: Improving service availability in the face of botnet attacks," in *Networked Systems Design and Implementation (NSDI)*, (Boston, MA), Apr. 2009.

[59] Harrison, W. S., Hanebutte, N., Oman, P., and Alves-Foss, J., "The MILS architecture for a secure global information grid," *Crosstalk: The Journal of Defense Software Engineering*, vol. 10, no. 10, 2005.

[60] Hartig, H., Hohmuth, M., Liedtke, J., Schonberg, S., and Wolter, J., "The performance of μ-kernel-based systems," in *ACM Symposium on Operating System Principles (SOSP)*, (Saint-Malo, France), Oct. 1997.

[61] Hofmeyr, S. A., Forrest, S., and Somayaji, A., "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[62] Honeynet Project, "Q8." `http://www.honeynet.org/papers/bots/`. Last accessed May 15, 2011.

[63] Hund, R., Holz, T., and Freiling, F. C., "Return-oriented rootkis: Bypassing kernel code integrity protection mechanisms," in *USENIX Security Symposium*, (Montreal, Canada), Aug 2009.

[64] IBM, "Ibm page detailer." `http://www.alphaworks.ibm.com/tech/pagedetailer/download`. Last accessed May 15, 2011.

[65] Intel, *System Programming Guide: Part 2*. Intel 64 and IA-32 Architectures Software Developer's Manual, 2004.

[66] Jiang, X. and Wang, X., "Out-of-the-box monitoring of VM-based high-interaction honeypots," in *Recent Advances in Intrusion Detection (RAID)*, (Surfers Paradise, Australia), Sept. 2007.

[67] Jiang, X., Wang, X., and Xu, D., "Stealthy malware detection through VMM-based 'out-of-the-box' semantic view," in *ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Nov. 2007.

[68] Johnson, M. K., "Introduction to linux." `http://tldp.org/HOWTO/INFO-SHEET-1.html`. Last accessed May 15, 2011.

[69] Jones, A. K. and Lu, Y., "Application intrusion detection using language library calls," in *Annual Computer Security Applications Conference (ACSAC)*, (New Orleans, LA), Dec. 2001.

[70] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Antfarm: Tracking processes in a virtual machine environment," in *USENIX Annual Technical Conference (ATC)*, (Boston, MA), June 2006.

[71] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "VMM-based hidden process detection and identification using Lycosid," in *ACM Virtual Execution Environment (VEE)*, (Seattle, WA), Mar. 2008.

[72] 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, Nov. 1991.

[73] Kasslin, K., "Evolution of kernel-mode malware." `http://igloo.engineeringforfun.com/malwares/Kimmo_Kasslin_Evolution_of_kernel_mode_malware_v2.pdf`. Last accessed May 15, 2011.

[74] Kephart, J. and Arnold, W., "Automatic extraction of computer virus signatures," in *Virus Bulletin*, (Jersey, Channel Islands, UK), 1994.

[75] King, S. T. and Chen, P. M., "Backtracking intrusions," in *ACM Symposium on Operating System Principles (SOSP)*, (Bolton Landing, NY), Oct. 2003.

[76] Kumar, S. and Spafford, E. H., "A pattern matching model for misuse intrusion detection," in *National Computer Security Conference*, (Baltimore, MD), Oct. 1994.

[77] KVM, "Kernel based virtual machine." `http://www.linux-kvm.org/page/Main_Page`. Last accessed May 15, 2011.

[78] Larry McVoy and Carl Staelin, "lmbench." `http://www.bitmover.com/lmbench/`. Last accessed May 15, 2011.

[79] Last, J. V., "Stuxnet versus the iranian nuclear program." `http://www.sfexaminer.com/opinion/op-eds/2010/12/stuxnet-versus-iranian-nuclear-program`. Last accessed May 15 2011.

[80] Le, C. H. H., "Protecting xen hypercalls: Intrusion detection/prevention in a virtualization environment." `https://circle.ubc.ca/handle/2429/14849`, 2009.

[81] Lindqvist, U. and Porras, P. A., "Detecting computer and network misuse through the production-based expert system toolset (p-best)," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 1999.

[82] Lindqvist, U. and Porras, P. A., "eXpert-BSM: A host-based intrusion detection solution for sun solaris," in *Annual Computer Security Applications Conference (ACSAC)*, (New Orleans, LA), Dec. 2001.

[83] Linn, C. and Debray, S., "Obfuscation of executable code to improve resistance to static disassembly," in *ACM Conference on Computer and Communications Security (CCS)*, (Washington, DC), Oct. 2003.

[84] Linn, C. M., Rajagopalan, M., Baker, S., Collberg, C., Debray, S. K., and Hartman, J. H., "Protecting against unexpected system calls," in *USENIX Security Symposium*, (Baltimore, MD), Aug. 2005.

[85] Lippmann, R. P., Fried, D. J., Graf, I., Haines, J. W., Kendall, K. R., McClung, D., Weber, D., Webster, S. E., Wyschogrod, D., Cunningham, R. K., and Zissman, M. A., "Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation," *DARPA Information Survivability Conference and Exposition*, vol. 2, 2000.

[86] Litty, L., Lagar-Cavilla, H. A., and Lie, D., "Hypervisor support for identifying covertly executing binaries," in *USENIX Security Symposium*, (San Jose, CA), Aug. 2008.

[87] LKCD Project, "LKCD - Linux Kernel Crash Dump." http://lkcd.sourceforge.net/. Last accessed May 15, 2011.

[88] Lu, L., Yegneswaran, V., Porras, P., and Lee, W., "Blade: An attack-agnostic approach for preventing drive-by malware infections," in *ACM Conference on Computer and Communications Security (CCS)*, (Chicago, IL), Oct. 2010.

[89] Marinescu, A., "Russian doll," in *Virus Bulletin Conference*, Aug. 2003.

[90] Marinescu, P. D. and Candea, G., "LFI: A practical and general library-level fault injector," in *IEEE/IFIP Conference on Dependable Systems & Networks (DSN)*, (Lisbon, Portugal), June 2009.

[91] Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., and Mitchell, J. C., "A layered architecture for detecting malicious behaviors," in *Recent Advances in Intrusion Detection (RAID)*, (Boston, MA), Sept. 2008.

[92] McAfee, "BackDoor-Rev.b." `http://vil.nai.com/vil/Content/v_136510.htm`. Last accessed May 15, 2011.

[93] McAfee, "McAfee Antivirus Products." `http://home.mcafee.com/store/free-antivirus-trials`. Last accessed May 15, 2011.

[94] McAfee, "Rootkits, part 1 of 3: The growing threat." `http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf`. Last accessed May 15, 2011.

[95] McCauley, E. J. and Drongowski, P. J., "KSOS: The design of a secure operating system," in *National Computer Conference*, (New York, NY), June 1979.

[96] Meng, J., Lu, X., and Dong, G., "A novel method for secure logging system call," in *Communications and Information Technology*, (Beijing, China), Oct. 2005.

[97] Mogul, J., Rashid, R., and Accetta, M., "The packet filter: An efficient mechanism for user-level network code," in *ACM Symposium on Operating Systems Principles (SOSP)*, (Austin, TX), Nov. 1987.

[98] MSDN, "Asynchronous procedure calls." `http://msdn.microsoft.com/en-us/library/ms681951(VS.85).aspx`. Last accessed May 15, 2011.

[99] Mutz, D., Robertson, W., Vigna, G., and Kemmerer, R., "Exploiting execution context for the detection of anomalous system calls," in *Recent Advances in Intrusion Detection (RAID)*, (Surfers Paradise, Australia), Sept. 2007.

[100] Necula, G. C., McPeak, S., Rahul, S., and Weimer, W., "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Conference on Compiler Construction (CC)*, (Grenoble, France), Apr. 2002.

[101] Newsome, J. and Song, D., "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Network and Distributed System Security Symposium (NDSS)*, (San Deigo, CA), Feb. 2005.

[102] Ning, P., Cui, Y., and Reeves, D. S., "Constructing attack scenarios through correlation of intrusion alerts," in *ACM Conference on Computer and Communications Security (CCS)*, (Washington, DC), Oct. 2002.

[103] Ning, P., Xu, D., Healey, C. G., and Amant, R. A. S., "Building attack scenarios through integration of complementary alert correlation methods," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2004.

[104] OffensiveComputing, "Storm worm process injection from the windows kernel." `http://www.offensivecomputing.net/?q=node/661`. Last accessed May 15, 2011.

[105] Packet Storm. `http://packetstormsecurity.org/UNIX/penetration/rootkits/{bdoor.c,blackhole.c,cheetah.c,server.c,ovas0n.c}`. Last accessed May 15, 2011.

[106] Packet Storm. `http://packetstormsecurity.org/0209-exploits/{apache-ssl-bug.c,apache-linux.txt}`. Last accessed May 15, 2011.

[107] Packet Storm, "All-root." `http://packetstormsecurity.org/UNIX/penetration/rootkits/all-root.c`. Last accessed May 15, 2011.

[108] Packet Storm, "Kaiten." `http://packetstormsecurity.org/irc/kaiten.c`. Last accessed May 15, 2011.

[109] Panda Security, "Panda Antivirus Pro 2011." `http://www.pandasecurity.com/usa/homeusers/solutions/antivirus/`. Last accessed May 15, 2011.

[110] Passmark Software, "PassMark Performance Test." `http://www.passmark.com/products/pt.htm`. Last accessed May 15, 2011.

[111] Paxson, V., "Bro: A system for detecting network intruders in real-time," in *Usenix Security Symposium*, (San Antonio, TA), Jan. 1998.

[112] Payne, B. D., Carbone, M., and Lee, W., "Secure and flexible monitoring of virtual machines," in *Annual Computer Security Applications Conference (ACSAC)*, (Miami, FL), Dec. 2007.

[113] Payne, B. D., Carbone, M., Sharif, M., and Lee, W., "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2008.

[114] Petroni, Jr., N. L., Fraser, T., Walters, A., and Arbaugh, W. A., "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *USENIX Security Symposium*, (Vancouver, BC, Canada), Aug. 2006.

[115] Petroni, Jr., N. L. and Hicks, M., "Automated detection of persistent kernel control-flow attacks," in *ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Nov. 2007.

[116] Popek, G. J. and Goldberg, R. P., "Formal requirements for virtualizable third generation architectures," in *ACM Symposium on Operating System Principles (SOSP)*, (New York, NY), Oct. 1973.

[117] Porras, P. A., Fong, M. W., and Valdes, A., "A mission-impact-based approach to infosec alarm correlation," in *Recent Advances in Intrusion Detection (RAID)*, (Zurich, Switzerland), Oct. 2002.

[118] Porras, P. A. and Neumann, P. G., "EMERALD: event monitoring enabling responses to anomalous live disturbances," in *National Computer Security Conference*, (Baltimore, MD), Oct. 1997.

[119] Pouget, F. and Dacier, M., "Alert correlation: Review of the state of the art," in *Research Report RR-03-093*, (Institut Eurecom, Sophia Antipolis, FRANCE), 2003.

[120] Provos, N., "Improving host security with system call policies," in *USENIX Security Symposium*, (Washington, DC), Aug. 2003.

[121] Qin, X. and Lee, W., "Statistical causality analysis of infosec alert data," in *Recent Advances in Intrusion Detection (RAID)*, (Pittsburgh, PA), Oct. 2003.

[122] Ramachandran, A., Mundada, Y., Tariq, M. B., , and Feamster, N., "Securing enterprise networks using traffic tainting," in *Technical Report GT-CS-09-15*, (Georgia Institute of Technology, Atlanta, GA), 2009.

[123] Rashid, F. Y., "Social engineering tops security flaw exploits as malware vector." `http://www.eweek.com/c/a/Security/Social-Engineering-Tops-Security-Flaw-Exploits-as-Malware-Vector-362042/`. Last accessed May 25, 2011.

[124] Richter, J., "Load your 32-bit DLL into another process's address space using injlib," *Microsoft Systems Journal*, vol. 9, May 1994.

[125] Rushby, J., "Design and verification of secure systems," in *ACM Symposium on Operating System Principles (SOSP)*, (Pacific Grove, CA), Dec. 1981.

[126] Rushby, J., "Proof of separability: A verification technique for a class of security kernels," in *Symposium on Programming)*, (Torino, Italy), Apr. 1982.

[127] Saydjari, O. S., Beckman, J. K., and Leaman, J. R., "LOCK Trek: Navigating uncharted space," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 1989.

[128] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P., "A fast automaton-based method for detecting anomalous program behaviors," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2001.

[129] Sekar, R., Cai, Y., and Segal, M., "A specification-based approach for building survivable systems," in *National Computer Security Conference*, (Washington, DC), Oct. 1998.

[130] Seshadri, A., Luk, M., Qu, N., and Perrig, A., "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *ACM Symposium on Operating Systems Principles (SOSP)*, (Stevenson, WA), Oct. 2007.

[131] Sharif, M., Lee, W., Cui, W., and Lanzi, A., "Secure in-vm monitoring using hardware virtualization," in *ACM Conference on Computer and Communications Security (CCS)*, (Chicago, IL), Nov. 2009.

[132] Sharif, M., Singh, K., Giffin, J., and Lee, W., "Understanding precision in host based intrusion detection: Formal analysis and practical models," in *Recent Advances in Intrusion Detection (RAID)*, (Surfers Paradise, Australia), Sept. 2007.

[133] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., and Kato, K., "BitVisor: A thin hypervisor for enforcing I/O device security," in *ACM VEE*, (Washington, DC), Mar. 2009.

[134] Shockley, W. R., Tao, T. F., and Thompson, M. F., "An overview of the gemsos class a1 technology and application experience," in *National Computer Security Conference*, Oct. 1988.

[135] Some Observations on Rootkits, "Microsoft Malware Protection Center." http://blogs.technet.com/b/mmpc/archive/2010/01/07/some-observations-on-rootkits.aspx. Last accessed May 15, 2011.

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

[137] Sophos, "Sophos security threat report 2010." `http://www.sophos.com/trmy10`. Last accessed May 15, 2011.

[138] Sourceforge, "Iperf." http://sourceforge.net/projects/iperf/. Last accessed May 15, 2011.

[139] Spafford, E., "Crisis and aftermath," *Communications of the ACM*, vol. 32, no. 2, 1989.

[140] Srivastava, A. and Giffin, J., "Tamper-resistant, application-aware blocking of malicious network connections," in *Recent Advances in Intrusion Detection (RAID)*, (Boston, MA), Sept. 2008.

[141] Srivastava, A. and Giffin, J., "Automatic discovery of parasitic malware," in *Recent Advances in Intrusion Detection (RAID)*, (Ottawa, Ontario), Sept. 2010.

[142] Srivastava, A., Lanzi, A., Giffin, J., and Balzarotti, D., "Operating system interface obfuscation and the revealing of hidde n operations," in *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, (Amsterdam, The Netherlands), July 2011.

[143] Srivastava, A., Singh, K., and Giffin, J., "Secure observation of kernel behavior," in *Technical Report GT-CS-08-01*, (Georgia Institute of Technology, Atlanta, GA), 2009.

[144] Staniford, S., Paxson, V., and Weaver, N., "How to 0wn the internet in your spare time," in *USENIX Security Symposium*, (San Francisco, CA), Aug. 2002.

[145] Swift, M. M., Bershad, B. N., and Levy, H. M., "Improving the reliability of commodity operating systems," in *ACM Symposium on Operating System Principles (SOSP)*, (Bolton Landing, NY), Oct. 2003.

[146] Symantec, "Windows rootkit overview." http://www.symantec.com/avcenter/reference/windows.rootkit.overview.pdf. Last accessed May 15, 2011.

[147] Szor, P. and Ferrie, P., "Hunting for metamorphic," in *Virus Bulletin Conference*, Sept. 2001.

[148] Ta-Min, R., Litty, L., and Lie, D., "Splitting interfaces: Making trust between applications and operating systems configurable," in *Operating Systems Design and Implementation (OSDI)*, (Seattle, WA), Oct. 2006.

[149] ThreatExpert, "Conficker/downadup: Memory injection model." http://blog.threatexpert.com/2009/01/confickerdownadup-memory-injection.html. Last accessed May 15, 2011.

[150] Tim Bray, "Bonnie." http://www.garloff.de/kurt/linux/bonnie. Last accessed May 15, 2011.

[151] UBRA, "Process hiding and the Linux scheduler," *Phrack*, vol. 11, Jan. 2005.

[152] UWE F. MAYER, "BYTEmark." `http://www.tux.org/~mayer/linux/bmark.html`. Last accessed May 15, 2011.

[153] VALDES, A. and SKINNER, K., "Probabilistic alert correlation," in *Recent Advances in Intrusion Detection (RAID)*, (Davis, CA), Oct. 2001.

[154] VENEMA, W., "Tcp wrapper: Network monitoring, access control and booby traps," in *USENIX UNIX Security Symposium*, (Baltimore, MD), Sept. 1992.

[155] VMWARE, "VMWare." `http://www.vmware.com/`. Last accessed May 15, 2011.

[156] WAGNER, D. and DEAN, D., "Intrusion detection via static analysis," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2001.

[157] WAHBE, R., LUCCO, S., ANDERSON, T. E., and GRAHAM, S. L., "Efficient software-based fault isolation," in *ACM Symposium on Operating System Principles (SOSP)*, (Asheville, NC), Dec. 1994.

[158] WALLACH, D. S. and FELTEN, E. W., "Understanding Java stack inspection," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 1998.

[159] WHITAKER, A., COX, R. S., SHAW, M., and GRIBBLE, S. D., "Constructing services with interposable virtual hardware," in *Networked Systems Design and Implementation (NSDI)*, (San Francisco, CA), Mar. 2004.

[160] WILLEMS, C., HOLZ, T., and FREILING, F., "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, Mar. 2007.

[161] WILLIAM D. NORCOTT, "IOzone." `http://www.iozone.org`. Last accessed May 15, 2011.

[162] WILLIAMS, D., REYNOLDS, P., WALSH, K., SIRER, E. G., and SCHNEIDER., F. B., "Device driver safety through a reference validation mechanism," in *Operating Systems Design and Implementation (OSDI)*, (San Diego, CA), Dec. 2008.

[163] WITCHEL, E., CATES, J., and ASANOVIC, K., "Mondrian memory protection," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (San Jose, CA), Oct. 2002.

[164] XENACCESS PROJECT, "XenAccess Library." `http://xenaccess.sourceforge.net/`. Last accessed May 15, 2011.

[165] Xiong, X., Tian, D., and Liu, P., "Practical protection of kernel integrity for commodity os from untrusted extensions," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2011.

[166] Xu, H., Du, W., and Chapin, S. J., "Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths," in *Recent Advances in Intrusion Detection (RAID)*, (Sophia Antipolis, France), Sept. 2004.

[167] Xu, M., Jiang, X., Sandhu, R., and Zhang, X., "Towards a VMM-based usage control framework for OS kernel integrity protection," in *ACM Symposium on Access Control Models and Technologies (SACMAT)*, (Sophia Antipolis, France), June 2007.

[168] Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N., "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2009.

[169] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E., "Panorama: Capturing system-wide information flow for malware detection and analysis," in *ACM Conference on Computer and Communications Security (CCS)*, (Arlington, VA), Oct. 2007.

[170] Zeng, Y., Hu, X., and Shin, K. G., "Detection of botnets using combined host- and network-level information," in *IEEE/IFIP Conference on Dependable Systems & Networks (DSN)*, (Chicago, Illinois), June 2010.

[171] Zorz, Z., "Operation aurora malware investigated." `http://www.net-security.org/malware_news.php?id=1223`. Last accessed May 15, 2011.