ABSTRACT

Title of dissertation:	PROPERTY-BASED INTEGRITY MONITORING OF OPERATING SYSTEM KERNELS
	Nick Louis Petroni, Jr. Doctor of Philosophy, 2008
Dissertation directed by:	Assistant Professor Michael Hicks Department of Computer Science

As the foundation of the trusted computing base, the operating system kernel is a valuable target for attackers of a computer system seeking maximum control and privilege. Furthermore, because the majority of modern security solutions rely on the correctness of at least some portion of the operating system kernel, skilled attackers who successfully infiltrate kernel memory can remain undetected indefinitely.

In this dissertation, we present an approach for detecting attacks against the kernel's integrity (commonly referred to as "rootkits"). Our approach, which we call *property-based integrity monitoring*, works by monitoring and analyzing the kernel's state at runtime. Unlike traditional security solutions, our monitor operates in isolation of, and independently from, the protected operating system and has direct access to the kernel's runtime state.

The basic strategy behind property-based monitoring is to identify a set of *properties* that are practical to check, yet are effective at detecting the types of changes an attacker might make — both known and yet-to-be-discovered. In this work, we describe a practical and effective property for detecting persistent control-flow modifications in running kernels, called state-based control-flow integrity (SBCFI). Furthermore, to address those data-only attacks that do not violate the kernel's control-flow, we introduce a high-level policy language system for enforcing *semantic integrity* constraints in runtime kernel data.

To evaluate the feasibility and effectiveness of our system, we have implemented two property-based integrity monitors for the Linux kernel — one using a virtual machine monitor and the other using a PCI-based coprocessor. We demonstrate that property-based monitoring is capable of detecting all publicly-available kernel integrity threats while imposing less than 1% overhead on the protected system. We conclude that property-based kernel integrity monitoring can be both practical and effective.

PROPERTY-BASED INTEGRITY MONITORING OF OPERATING SYSTEM KERNELS

by

Nick Louis Petroni, Jr.

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Acknowledgements

There are many people without whom the completion of this dissertation would not have been possible. First, I would like to thank my advisors, Bill Arbaugh and Mike Hicks, both of whom have provided me with an incredible amount of support, mentorship, opportunity, education, and friendship. While I can say without hesitation that they are two of the busiest and hardest-working people I have ever met, I cannot recall an occasion where either one failed to make time for me or my work. In addition, I cannot thank them enough for their constant flexibility, particularly for allowing me to work remotely for such an extended period of time and to take a few breaks from research to pursue other opportunities. If I have learned anything about research, management, systems, or security, it is because of Bill and Mike.

I would also like to thank the other members of my committee — Jeff Foster, Jeff Hollingsworth, and Gang Qu — for their time and valuable feedback. I am particularly thankful to Jeff F., who was consistently willing to provide quick and helpful comments and guidance over the last couple of years. Finally, I would like to thank Professor Jesus Izaguirre who, through his support and encouragement, led me to graduate school in the first place.

In addition to my advisors, I have learned much from my collaborators and colleagues at UMCP. Over the last six years, members of the MISSL lab and language research group have been the backbone of my academic progress. I am constantly impressed with the quality of research produced by these individuals and, more importantly, the quality of their work ethic and character. I am particularly grateful to Tim Fraser, with whom I have had the pleasure to work on several occasions during the last four years. Tim was one of the driving forces behind (and co-inventors of) Copilot, described in Chapter 4 of this dissertation. He also helped flesh out much of the rule-based system described in Chapter 6, including the development of a number of proof-of-concept attack vectors. I have benefited greatly from Tim's knowledge and experience in OS security and, more importantly, from his willingness to guide, review, and help correct my work.

In my many hours of fun outside of UMCP, I have had the great privilege to work extensively with Mike Shields over the last few years. Mike and I have spent endless hours hacking in the lab (typically into the early hours of the morning), discussing work on the phone, and iterating via email over any number of issues. I am grateful to Mike for all of his friendship, guidance, mentorship, and support. Mike is a true renaissance man, and I have learned much from him about work, life, sports, business, family, service, and faith.

I would like to thank Zach Schafer and AAron Walters, two of my best friends and truly two of the best people in the world. Without their help and support, I am quite certain that I could not have completed my degree. For more than ten years, I have looked up to Zach and AAron, in spite of the fact that both are actually a few months younger than me. In many ways, I have followed in Zach's footsteps as we have encountered the same milestones along life's path. At each transition, he has been a shining example of how to live with faithfulness, compassion, laughter, and humility. While he is fond of claiming otherwise, Zach is much smarter than me. He is also a much better athlete than me and is an amazing friend, husband, brother, son, and father.

No person has been a stronger friend to me, or advocate for me, than AAron Walters. AAron and I have spent countless hours (including far too many all-nighters) hacking, researching, building systems, and talking about life. Like Zach, AAron is better than me at just about everything he does. He is defined by his dedication, work ethic, and, most importantly, his faith. We have collaborated on a number of projects and AAron has contributed greatly to the ideas and methodologies that inspired this dissertation. In addition to being a great sounding board and excellent engineer, he is the most loyal of friends.

My parents have made every sacrifice to provide my sisters and I with the best possible lifestyle. If we have achieved anything, or will in the future, it is only due to their generosity, hard work, faithfulness, and dedication. I am particularly thankful for the gifts of faith and education, the latter of which they have given to each of us at great cost to their own comfort and lifestyle. More importantly, I thank my parents, my sisters, Jason, and Neal for their unconditional love and constant support. My two sisters, Stephanie and Alyson, are truly two of the most amazing people I have ever met. They are both beautiful, smart, funny, kind, and generous, and I will never be able to thank them enough for all that they do. Together with my grandparents — Alyse, Lewis, Rocco, and Jay — and all of my aunts, uncles, and cousins, they have taught me the true meaning of love and the value of family. I would also like to thank my new family — Tim, Joanne, Betsy, Maddie, Stuart, Mike, Lisa, Arya, Meaghan, Mom, Dad, and all of the Bovich clan — for accepting me and loving me as one of their own.

I am eternally grateful to my amazing wife, Colleen, for her constant love and support. I am more proud of her than she will ever know, and I could not have endured graduate school without her help and patience. At all times she has been my rock and I love her dearly.

Finally, above all, I thank God for the many blessings in my life and for the gift of life itself.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABR	Weighted Aggregate Byte Rate
API	Application Programming Interface
AVC	SELinux Access Vector Cache
BFS	Breadth First Search
CFG	Control-flow Graph
CFI	Control-flow Integrity
CIL	C Intermediate Language
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
FBI	Federal Bureau of Investigation
GC	Garbage Collector
HTTP	Hypertext Transport Protocol
IDT	Interrupt Descriptor Table
I/O	Input / Output
IOMMU	I/O Memory Management Unit
LKM	Loadable Kernel Module
MMU	Memory Management Unit
OS	Operating System
PCI	Peripheral Component Interconnect
PID	Process Identifier
RAM	Random Access Memory (main system memory)
RISC	Reduced Instruction Set Computer
SBCFI	State-based Control-flow Integrity
SELinux	Security Enhanced Linux
SID	SELinux Security Identifier
SIQR	Semi-Interquartile Range
SMM	System Management Mode
SMP	Symmetric Multi-Processor
SPEC	Standard Performance Evaluation Corporation
TCB	Trusted Computing Base
TCG	Trusted Computing Group
UID	User Identifier
VFS	Virtual Filesystem
VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor

Chapter 1

Introduction

As society continues to expand its dependence on computer systems for all aspects of life, the trustworthiness of those systems remains a vital concern. Increasingly, we rely on computers to manage our personal data, provide a platform for interpersonal communication, and control critical infrastructure, such as power distribution, communication networks, and transportation systems. Because of this heightened dependence, we would like to have confidence that our computing infrastructure will operate as expected, remain safe, and resist attempted attacks from those with nefarious objectives.

Unfortunately, obtaining confidence in real-world systems is a difficult problem, particularly in the face of malicious adversaries. Modern systems, especially those connected to the public Internet, face an almost continual barrage of attempted attacks with varying degrees of sophistication and success. A recent survey conducted by the Federal Bureau of Investigation (FBI) showed that 87% of surveyed organizations had a computer security incident in the previous 12 months [79]. More alarmingly, these attacks are often successful, giving unauthorized parties control of machines without the knowledge of the system's rightful owner or administrator. The same FBI survey indicated that between 13% and 37% of organizations experienced an incident that resulted in unauthorized access [79].

System compromise is frequently the result of adversaries exploiting design or implementation flaws in software or hardware. Attackers may also exploit system misconfigurations or misplaced trust, e.g., granted to an insider. Regardless of the mechanism employed for system entry, an attacker may use the system to achieve any number of objectives once he or she has gained control. In some instances, the adversary's goal might be specific and short-lived. For example, the attacker may simply desire access to files or information stored on the machine. Alternatively, he or she may be in need of a "stepping stone" — an unrelated third party's system that the attacker uses to provide temporary anonymity when attacking other victims [108]. At the other extreme, an adversary may have a much more long-term set of goals in mind. For example, specific machines may be the target of espionage in the form of keystroke logging or packet sniffing. More recently, the Internet has seen the rise of massive *botnets* — hundreds of thousands of compromised machines operating under the remote command and control of attackers without the knowledge of their rightful owners [66, 23]. Attackers use these large "robot networks" to commit fraud, send unwanted email solicitations, or launch large-scale denial of service attacks [80]. Regardless of the specific objectives held by a particular attacker, most attacks have one fundamental goal in common: they must remain undetected.

Early attempts to prevent detection took the form of "trojan horse" attacks, whereby attackers replace one or more system files, such as system administrator tools, shared libraries, or security applications, in order to filter the normal output of the replaced component [6]. The advent of filesystem integrity checkers, such as Tripwire [56], led attackers to develop techniques for modifying code once it has been loaded, e.g., through DLL injection attacks [67]. Attackers also turned to making modifications to the operating system (OS) kernel. As the lowest software layer and the largest provider of system services, the operating system kernel is of critical importance for the integrity of the system as a whole. Because the kernel controls access to all resources, even unmodified versions of system administrator tools will report inaccurate information if they rely on a compromised kernel. Traditionally, attackers have used one or more of these mechanisms collectively to form toolkits, commonly referred to as “rootkits.” The term comes from the toolkit’s ability to help maintain superuser privileges (called “root” on UNIX-based platforms) by obfuscating the attacker’s presence from rightful system administrators. Recently, kernel-level rootkit techniques have been used in a growing number of attacks, such as the Storm botnet worm [100].

In this dissertation, we present a system for detecting attacks against operating system kernels. There are a number of challenges in trying to identify kernel-level compromises. First, in most systems, the kernel operates at the highest level of privilege. Attackers that gain access to the kernel have the ability to modify any software installed on the system. As a result, security mechanisms that were once considered protected, such as trusted kernel modules, will also be susceptible to attacker control. Second, kernels typically have strict performance requirements. Kernels handle all system interrupts and service a highly multitasked system. Any security solution that is added to the system must not significantly degrade the kernel’s performance. Finally, any attempt to detect intrusions must have a viable approach for distinguishing “good” from “bad.” There are many proposed methods for performing intrusion detection in the literature. Choosing an approach that is likely to detect current and future attack vectors, yet is practical to implement and manage, is a difficult task.

Our approach is summarized as follows:

1. The most effective way to monitor a kernel's integrity is to determine when it is behaving correctly (or, conversely, detect when it is not). Because this would require a full specification of correct behavior, which is intractable, we instead propose to monitor whether the kernel's behavior exhibits certain *properties* that, if violated, indicate that the kernel is not behaving correctly. We aim to choose a set of properties that are both practical to check and likely to be violated by real attacks.

2. To detect when the kernel has violated a relevant property, we introduce a monitor that analyzes the kernel's state as it executes. The monitor implements a set of checks that validate the kernel's state with regard to the chosen properties.

3. To address the issues of protecting the monitor and reducing imposed overhead, we utilize an isolated, independent monitor mechanism, such as a trusted piece of hardware or protected software environment. Rather than impede the kernel with in-line checks, we allow the unmodified kernel to operate as usual. An external monitor with access to the kernel directly analyzes its state asynchronously, checking for violations of the specified runtime properties.

Figure 1.1 shows the high-level components of our monitor infrastructure. The

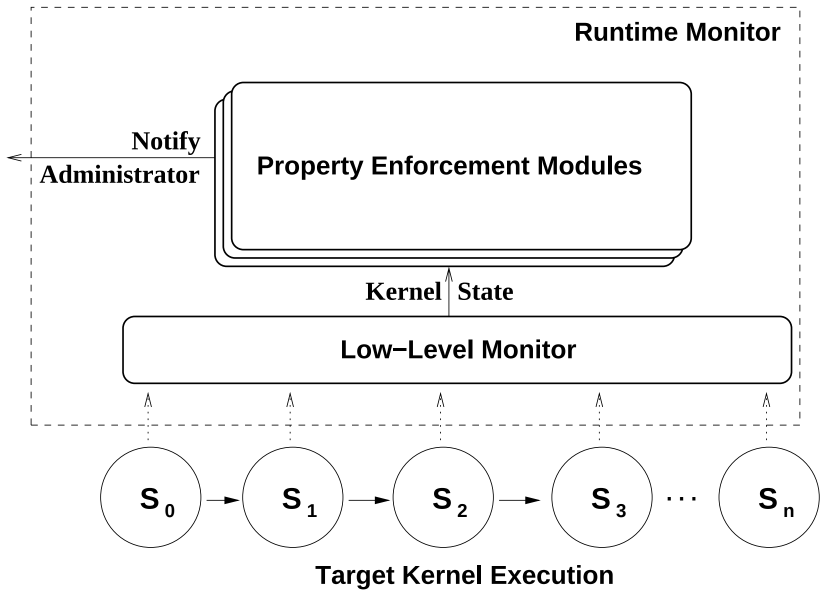

Figure 1.1: Property-based kernel monitor.

foundation of the system is a trusted low-level monitor that operates independently from the running kernel, but has access to all of its state, such as memory and CPU registers. We have implemented two monitor mechanisms – a coprocessor-based PCI add-in card and a software solution based on modern virtualization technology. The low-level monitor provides relevant kernel state, such as processor registers and RAM, to a property enforcement engine, which performs property verification checks on the kernel's runtime state.

The enforcement engine is comprised of a set of property enforcement modules, each of which is responsible for checking one or more kernel properties. In this dissertation, we describe two property enforcement modules. The first, which enforces a novel property called *state-based control-flow integrity*, ensures that the kernel has not been modified to execute unauthorized code paths. The second allows experts to manually describe additional properties of the kernel's state in a high-level specification language. When used in combination, we have demonstrated that these two modules are capable of detecting all publicly-available Linux kernel rootkits. The resulting system minimally impacts the performance of the protected system – less than 1% when operated in a realistic configuration.

1.1 Contributions

In summary, this dissertation makes the following contributions:

1. Our primary contribution is to show that it is possible to build a practical, effective *property-based monitor* for operating system kernels that is capable of detecting malicious modifications made by attackers.

2. We perform an analysis of a collection of real-world kernel integrity attacks and, based on this analysis, provide a classification of the types of modifications an attacker can make to the kernel's state. We further show how this classification can be used to devise a reasonable set of *properties* for monitoring the kernel's execution (Chapter 2).

3. We introduce a monitor architecture that can easily implement property verification procedures capable of enforcing relevant properties of the kernel's state (Chapter 3).

4. We outline the requirements for a low-level state-based monitor and describe the design and implementation of two such monitors – one coprocessor-based

and one virtual machine-based (Chapter 4).

5. We identify control-flow integrity (CFI) as a critical property for kernel integrity and develop a practical approximation, called state-based control-flow integrity (SBCFI), that can effectively be applied to real-world kernels using a property-based monitor (Chapter 5).

6. We propose a specification language-based system for enforcing security-relevant properties on kernel non-control data (Chapter 6).

7. We implement a prototype of the designed system for monitoring the Linux kernel and demonstrate its effectiveness at detecting all publicly-available kernel-level rootkits for the Linux 2.4 kernel with minimal (less than 1%) impact on the protected system.

As we describe in Chapter 7, most other approaches are far too impractical due to their high overhead or unrealistic requirements. At the other extreme, some approaches have more reasonable operational requirements, but fail to detect a wide range of attacks. Based on our analysis and experiments, we believe that our architecture strikes an important balance as the most effective, yet most practical system proposed to date.

Chapter 2

Kernel Integrity Threats

In this chapter, we describe how attackers can modify the kernel at runtime and why they do so. Based on our analysis of a set of publicly-available kernel threats for the Linux kernel, we explain various ways that an attacker might modify an OS kernel. Using this data, we divide the set of possible changes into *control-flow modifying* and *non-control-flow modifying*, which is useful for developing a set of runtime properties that can be used to effectively detect classes of attacks, both known and unknown.

2.1 Taxonomy

Rootkits have been categorized in a number of ways in the literature [36]. In this section, we summarize our analysis of 25 rootkits that we have obtained for the Linux kernel. We classify each threat with regard to two separate characteristics — functionality (the set of features provided by the rootkit) and mechanism (the types of changes made to the kernel). While the former is useful for understanding the motives and intentions of attackers, and is therefore helpful for predicting future attack vectors, the latter is essential for those attempting to prevent and detect attacks. We begin by summarizing the features provided by our analyzed rootkits. Then we describe the various techniques used to implement these features.

2.1.1 Functionality

Table 2.1 contains the list of Linux kernel rootkits that we were able to obtain.[1] The left section of the table presents the set of attack goals that we identified for each analyzed rootkit. Based on our analysis, the objectives of each rootkit fall into one or more of the following categories: user-space object hiding (HID), privilege escalation (PE), reentry/backdoor (REE), reconnaissance (REC), and defense neutralization (NEU). We discuss each category in more detail.

Hiding refers to the concealment or masking of information contained in the kernel that may indicate the presence of the attacker. The concealed information typically relates to resources currently in use by the attacker, such as files, network connections, or running processes. Kernel-level objects, including modules or kernel threads, may also be hidden.

In addition to hiding, many rootkits implement a mechanism for providing ***privilege escalation*** for malicious userspace processes that are controlled by the attacker. As the term "rootkit" suggests, one popular technique is to elevate privilege by changing the effective user identifier of a process to that of "root" or "administrator," thereby giving that process access to nearly all system resources.[2] Depending on the access control mechanisms employed by the kernel, other changes might also be made. For example, attackers may add specific *capabilities* to a process, change entries in an *access control list*, or compromise some other form of

[1] All of the rootkits used in this work were obtained from public web sites. Many came from `http://packetstormsecurity.org/`.

[2] Note that this attack assumes traditional discretionary access control, where access is granted on a per-user basis, as implemented by the vast majority of current deployments.

Attack Name	**Functionality**					**Modifications**			
						Control-flow			
	HID	**PE**	**REE**	**REC**	**NEU**	**TXT**	**REG**	**FP**	**NCD**
Adore	X	X						P	P
Adore-ng	X	X			X			P	P
All-root		X						P	
Anti Anti Sniffer	X			X				P	
enyelkm	X	X	X			P			
hp	X	X							P
kdb	X	X						P	
KIS	X		X					P	P
Knark	X	X	X					P	P
Linspy			X					P	
Maxty				X				P	
Modhide	X							P	P
Mood-nt	X			X		B	P	B	
Override	X	X						P	
Phantasmagoria	X					P			P
Phide	X							P	
Rial	X							P	
Rkit		X						P	P
Taskigt		X						P	
Shtroj2		X						P	
SucKIT	X			X		P		T	
SucKIT2	X	X	X	X				B	
Superkit	X		X	X		P		T	
Synapsys	X							P	
THC Backdoor		X						P	

Table 2.1: Summary of analyzed kernel threats.

security policy. In Section 6.2, we describe a simulated attack that we have implemented against the Security Enhanced Linux (SELinux) access vector cache (AVC). This attack is another example of kernel-level privilege escalation.

Frequently, attackers insert some form of "backdoor" – an alternate access path used to reenter the system at a later time that typically bypasses authentication and authorization checks. Backdoors are useful to attackers that lose their original entry path, for example, through the patching of an existing vulnerability or the disabling of a compromised account. More generally, attackers may be interested in installing and configuring mechanisms for directing future actions or workloads to be performed by the compromised machine. We refer to such "command and control" channels collectively as ***reentry*** mechanisms. As an example of a kernel-level backdoor, consider `enyelkm`, a Linux kernel rootkit that initiates a root shell connection to a remote host when carefully crafted ICMP packets are received by the infected kernel. Similar remote shell functionality was found in 6 of our 25 analyzed threats.

Another feature found in many attacks is ***reconnaissance***, which refers to functionality that allows attackers to gather information from a target system. Typical examples of reconnaissance include keystroke logging, packet sniffing, collecting user information (including passwords or other credentials), and file exfiltration. Several of our analyzed rootkits implemented kernel-level reconnaissance, although many others provide features, such as hiding and privilege escalation, that facilitate the stealthy use of userspace reconnaissance components.

The final major category of capabilities encountered during our rootkit analysis is ***defense neutralization***. Defense neutralization mechanisms are used to effectively disable, in whole or in part, various security and safety features of the target system. As shown in Table 2.1, only one analyzed rootkit, `Adore-ng`, provides a feature that we classify as defense neutralization. However, features such as hiding and reentry can be viewed as forms of defense neutralization, because they are frequently used to prevent standard security solutions from accessing their intended targets. As an example, consider a set of malicious files that have been hidden from all processes, except those of the attacker. These files will be inaccessible to antivirus and filesystem integrity checkers, thereby thwarting detection. More direct attacks against system defenses are also possible, such as the elimination of kernel firewall rules or the disabling of encryption capabilities [8].

2.1.2 Mechanism

Having described the capabilities found in our attack corpus, we now explain in more detail how an attacker can introduce these features by modifying a target kernel. While the specifics of each attack (and of each kernel implementation) can vary significantly, we have generalized the techniques found in all of our rootkit samples into a few categories. Specifically, we characterize the kernel modifications made by each rootkit with regard to two axes: (1) the type of object modified and (2) the duration over which the change remains present in kernel memory. These two characteristics will help us devise a useful set of properties for monitoring a subset of the kernel's state.

As shown in the right section of Table 2.1, we classify the type of object modified as falling into one of the following categories: (1) the kernel's machine code (also called simply "code" or "text" (TXT)), (2) processor registers (REG), (3) kernel control-flow data, such as function pointers (FP), and (4) kernel non-control data (NCD).

As shown in the left side of Figure 2.1, kernel text modifications refer to a class of changes whereby, at runtime, an attacker overwrites a portion of the kernel's

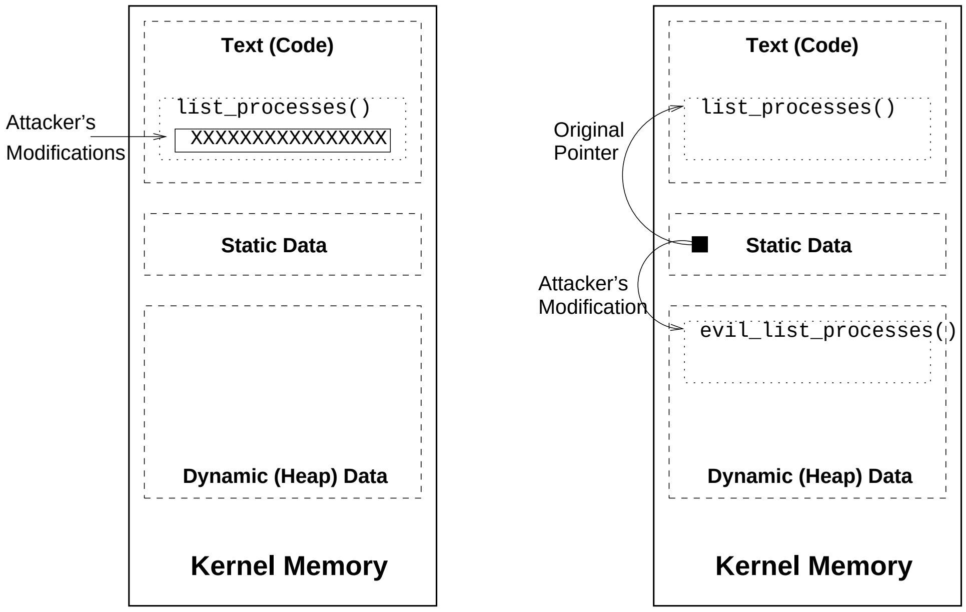

Figure 2.1: Control-flow manipulation by rewriting kernel code (left) or function pointers (right).

machine instructions to change its functionality. For example, an attacker may add logic to an instruction sequence to insert or remove conditionals, insert direct jumps to other code found elsewhere in memory, or simply overwrite text with `nop` instructions, effectively "whiting out" specific functionality. As shown in Table 2.1, a handful of analyzed rootkits were found to modify the kernel's in-memory code. The remainder chose to change the kernel's execution by targeting registers or some other memory-resident object.

Most CPUs contain two types of registers – those used to perform calculations or data manipulations and those used to control the processor's execution. Attackers that are able to modify the latter can manipulate kernel or userspace execution in arbitrary ways. For example, on Intel x86 processors, registers controlling the current code segment (`cs`) and the current instruction pointer (`eip`) determine the immediate set of code executing on a processor at a given time. In addition, several registers control the set of instructions that might be executed as a result of specific events — including function returns (`esp`), hardware interrupts and software exceptions (`idtr`), system calls (`sysenter`), and debug breakpoints (`dr0-dr3`). Attackers that manipulate these registers can cause their unauthorized code to be executed in addition to, or instead of, real kernel code. One of our analyzed rootkits, `Mood-NT`, uses the x86 debug registers to trap reads that might detect modifications made to other parts of memory.

Attackers may also influence the kernel's execution by manipulating data values in memory that are used to calculate dynamic branches at runtime. Traditional buffer overflow attacks, such as stack-smashing, represent a transient form of control-data modifications. In these attacks, return addresses or other control-related data are overwritten such that dynamic branches (e.g., `ret` instructions) target the wrong memory location. As shown in Table 2.1, 22 of our analyzed rootkits (88%) utilized a control-data modification attack. In all cases, these modifications were found to take the form of modified kernel function pointers. Many modern kernels utilize dynamic call targets, in the form of C language function pointers, in order to facilitate modularity and object-oriented design. While some of these function pointers exist in the statically allocated part of kernel data, the majority are found within the many dynamic data structures allocated in the kernel's heap.

Rather than attempt to alter the set of instructions executed by the kernel, attackers may be able to achieve their objectives by simply modifying the kernel's non-control data, thereby using built-in functionality in unexpected, surreptitious ways. As shown in Table 2.1, eight of our analyzed rootkits modify kernel non-control data. In many cases, these modifications amount to simple value changes to critical kernel identifiers such as process ids and user ids. As described previously, changing a process's user identifier is an effective mechanism for modifying that process's privileges in many modern kernels. More complex data changes can also be made. As we describe in detail in Section 2.2.3, one of our analyzed rootkits, `hp`, effectively hides processes by modifying kernel non-control data pointers.

In addition to the type of the modified object, malicious modifications may be classified as either *transient* or *persistent*. Informally, transient modifications are those for which the unauthorized modification is limited to a single (or small set) of executing threads for a short period of time. For example, the `SucKIT` rootkit temporarily redirects a system call handler to the kernel's `kmalloc()` function just long enough to use the modified system call to allocate a kernel data buffer. Once this action is complete, the system call is returned to its original value. In contrast, persistent modifications refer to extended periods of altered execution.[3] Several examples of persistent modifications are described later in this chapter. During our analysis, we found instances of both persistent and transient changes. For each modification that we identified in columns 7–10 of Table 2.1, we also determined whether the modification is persistent, represented by a P, or transient, represented by a T. Columns with a B indicate that both persistent and transient modifications

[3]The word persistent, when applied to malicious software, is sometimes used to describe malware that remains active (or "survives") after a machine has been rebooted or even reinstalled. In contrast, we use the term persistent to describe specific changes made by malicious software that remain observable for extended periods of execution, prior to a reboot.

of a given type are made. We found that all of the analyzed threats make some persistent change to the kernel.

2.2 Example: Process Hiding in Linux

Having described the various goals and mechanisms found in our analyzed rootkits, we now present an example of how, given a particular objective, an attacker can utilize one or more of the presented techniques to achieve that objective. The premise underlying this example is that the attacker has one or more running processes that he or she wishes to hide from system administrators, security software, or other userspace processes that are not controlled by the attacker. While evidence of the existence of a process can be found in many places,[4] the first step is to filter simple queries that contain lists of running processes. We begin by describing the typical mechanism by which userspace code retrieves the list of processes from the running kernel. We then explain how text, registers, and function pointers can be modified to insert malicious kernel code capable of hiding userspace processes. Finally, we describe a non-control data attack for achieving the same.

2.2.1 Linux Kernel Process Accounting

To understand the specific mechanisms available to attackers seeking to hide userspace processes, we first provide a (slightly simplified) explanation of the typical means by which the set of running processes is communicated from the kernel to

[4]In fact, some rootkit detection systems rely on the existence of various system artifacts to detect hidden processes [31].

userspace processes. Readers familiar with the internals of Linux process accounting may wish to skip this subsection.

In the Linux kernel, the primary mechanism for communicating process metadata across the user/kernel boundary is the process information pseudo-filesystem (or, simply "proc filesystem"). Typically mounted at `/proc`, the proc filesystem is a hierarchical interface for providing information about currently allocated userspace processes and kernel threads. In Linux, each process is assigned a unique name (integer), known as a process identifier (PID). The proc filesystem implements directories for each PID, each of which contains a set of virtual directories and files that can be read (and in some cases written) to obtain process information. When a process needs to know the list of currently running processes, it simply performs a directory listing of `/proc`. The resulting set of numerical subdirectory names represents an exhaustive list of running processes. System administration tools, such as `ps`, are typically implemented as a series of proc filesystem look-up and read operations.

When a process requires work from the kernel, such as performing a directory listing, it makes a request via the kernel's system call interface. In x86 Linux, system calls are initiated via the software interrupt instruction `int` with a parameter of 0x80.[5] As shown by the lines labeled "1" in Figure 2.2, the numerical parameter identifies which interrupt handler should be executed by providing an index to the processor's interrupt descriptor table (IDT). Because the kernel supports multiple system calls, the top-level system call handler is passed, usually in a processor

[5]On Pentium II (or later) processors, the "fast system call" `sysenter` instruction may also be used to initiate the system call handler.

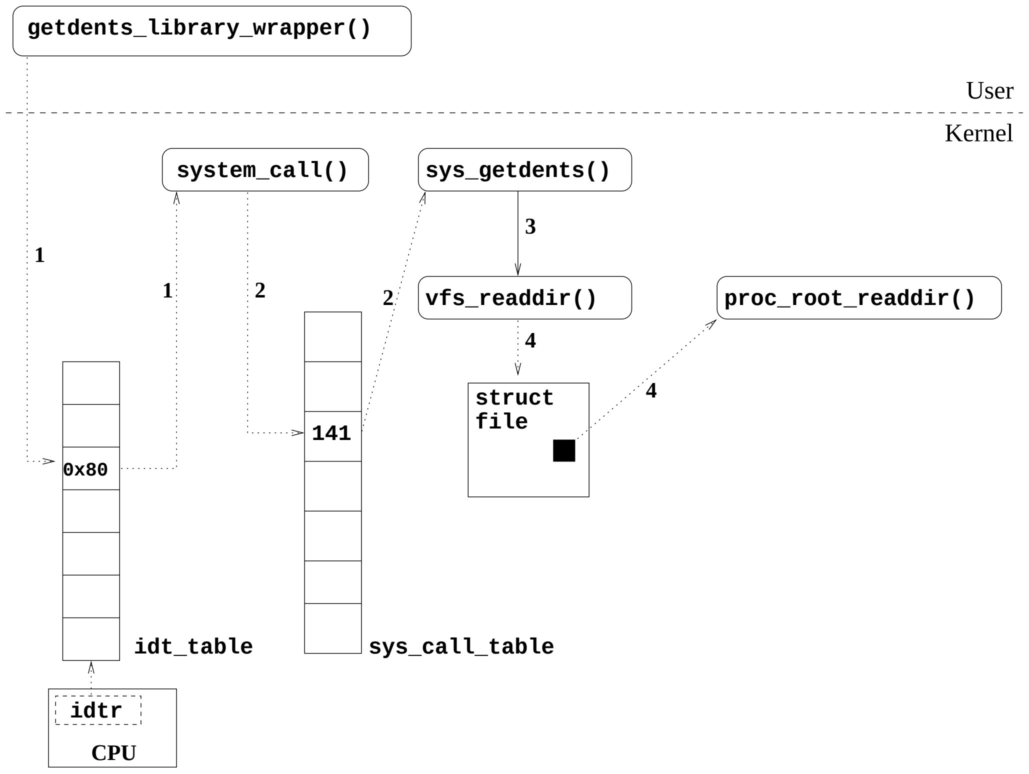

Figure 2.2: Linux process listing data relationships and control-flow.

register, at least one additional parameter — an integer that indicates the system call number to be executed. Using this number as an index into a table of function pointers (referred to as the "system call table"), a specific system call handler is then called. In the case of a directory listing, the `sys_getdents()` or `sys_getdents64()` system call is used, as shown by the lines labeled "2" in Figure 2.2.

After performing a set of access checks and resolving the passed file descriptor (obtained using an earlier `open()` system call), `sys_getdents()` performs a direct call (labeled "3" in Figure 2.2) to `vfs_readdir()` in the kernel's virtual filesystem (VFS) infrastructure. The VFS is a subsystem that provides a generic interface for executing various operations, including directory listing, for an extensible set of filesystem implementations. The proc pseudo-filesystem is one such implementation. Each implemented filesystem provides a set of callback functions, which are passed to the VFS at module initialization. As shown by the lines labeled “4” in Figure 2.2, `vfs_readdir()` calls, via a function pointer, the particular `readdir()` function that has been associated with the corresponding `file` object at an earlier time (e.g., when the file was opened). A common set of data structures are used to pass the directory information back to userspace, regardless of the low-level implementation.

In the case of proc, directory reads for the root directory are performed by the `proc_root_readdir()` function. This function identifies the list of all active processes and returns a corresponding directory entry for each PID, along with a few non-numeric entries. In order to determine the set of current PIDs, the kernel must reference its internal process accounting infrastructure. We now describe the details of Linux's process accounting subsystem.

The primary data structure for process management in the Linux kernel is the `task_struct` structure [64]. All threads are represented by a `task_struct` instance within the kernel. A single-threaded process will therefore be represented internally by exactly one `task_struct`. Since scheduling occurs on a per-thread basis, a multi-threaded processes is simply a group of `task_struct` objects that share certain resources such as memory regions and open files, as well as a few other properties, including a common process identifier.

In a correctly-running system, all `task_struct` objects are connected in a complex set of linked lists that represent various groupings relevant to that task at a particular time. For accounting purposes, all tasks are members of a single doubly-linked list, identified by the `task_struct.tasks` member. This list, which we refer to as the all-tasks list, insures that any kernel function needing access to all tasks, such as `proc_root_readdir()`, can easily traverse the list and encounter each task exactly once. The head of the task list is the `swapper` process (PID 0), identified by the static symbol `init_task`. To support efficient lookup based on PID, the kernel also maintains a hash table that is keyed by PID and whose members are hash-list nodes located in the `task_struct.pid` structure. Only one thread per matching hash of the PID is a member of the hash table; the rest are linked in a list as part of the `task_struct.pid` member. Other list memberships include parent/child and sibling relationships and a set of scheduler-related lists discussed next.

Scheduling in the Linux kernel is also governed by a set of lists. Each task exists in exactly one state. For example, a task may be actively running on the processor, waiting to be run on the processor, waiting for some other event to occur (such as I/O), or waiting to be cleaned up by a parent process. Depending on the state of a task, that task will be a member of at least one scheduling list somewhere in the kernel. At any given time, a typical active task will either be a member of one of the many wait queues spread throughout the kernel or a member of a per-processor run queue. Tasks cannot be on both a wait queue and a run queue at the same time.

Given this summary of some of the Linux kernel's implementation details, we now describe a few methods by which an attacker can achieve the previously stated goal – preventing a running process from being reported to other processes.

2.2.2 Control-Flow Modifications for Process Hiding

Attackers with access to kernel memory can hide a process in a number of ways. One obvious approach is simply to change the kernel's execution along the described path so that information about the surreptitious processes is removed from (or never included in) the returned data. This can occur as a result of any of the following actions:

- The attacker modifies the `idtr` or `sysenter` registers, redirecting the top-level system call handler. The new system call handler calls a modified version of `sys_getdents()`, but the correct version of all other system calls.
- The attacker modifies the in-memory interrupt descriptor table (rather than the register itself) to reference a new top-level system call handler (Figure 2.2, label "1").
- The attacker directly rewrites the text of the top-level system call handler or one of the other functions called (e.g., `sys_getdents()`, `vfs_readdir()`, or `proc_root_readdir()`).
- The attacker modifies the system call table to redirect the call to `sys_getdents()` to an injected function (Figure 2.2, label "2").
- The attacker modifies the function pointer referenced via the `file` structure for the open `/proc` directory entry, thereby redirecting the call to `proc_root_readdir()` to an injected function (Figure 2.2, label "4").

2.2.3 Non-Control Data Process Hiding

Attackers do not need to alter the code that the kernel executes to hide processes within a running kernel. In fact, they do not need to rely on manipulating the control flow of the kernel at all. Instead, adversaries have found techniques to hide their processes even from correct, unmodified kernel code. By directly manipulating the underlying data structures used for process accounting, an attacker can quickly and effectively remove any desired process from the view of standard, unmodified administrator tools. While the process remains hidden for accounting purposes, it continues to execute normally and will remain unaffected from the perspective of the scheduler. This technique is implemented by the `hp` rootkit, the only analyzed rootkit that made no kernel control-flow modifications.

Figure 2.3 depicts the primary step of the attack: removing the process from the doubly-linked all-tasks list (indicated by the solid line between tasks). Since this list is used for all process accounting functions, such as the `proc_root_readdir()` call in the `/proc` filesystem, removal from this list provides all of the stealth needed by an adversary. For an attacker who has already gained access to kernel memory, making this modification is as simple as modifying two pointers per hidden process. As a secondary step to the attack, adversaries might also choose to remove their processes from the PID hash table (not pictured) to prevent the receipt of unwanted signals.

As shown in Figure 2.3, a task not present in the all-tasks list can continue to function because the set of lists used for scheduling is disjoint from the set used for

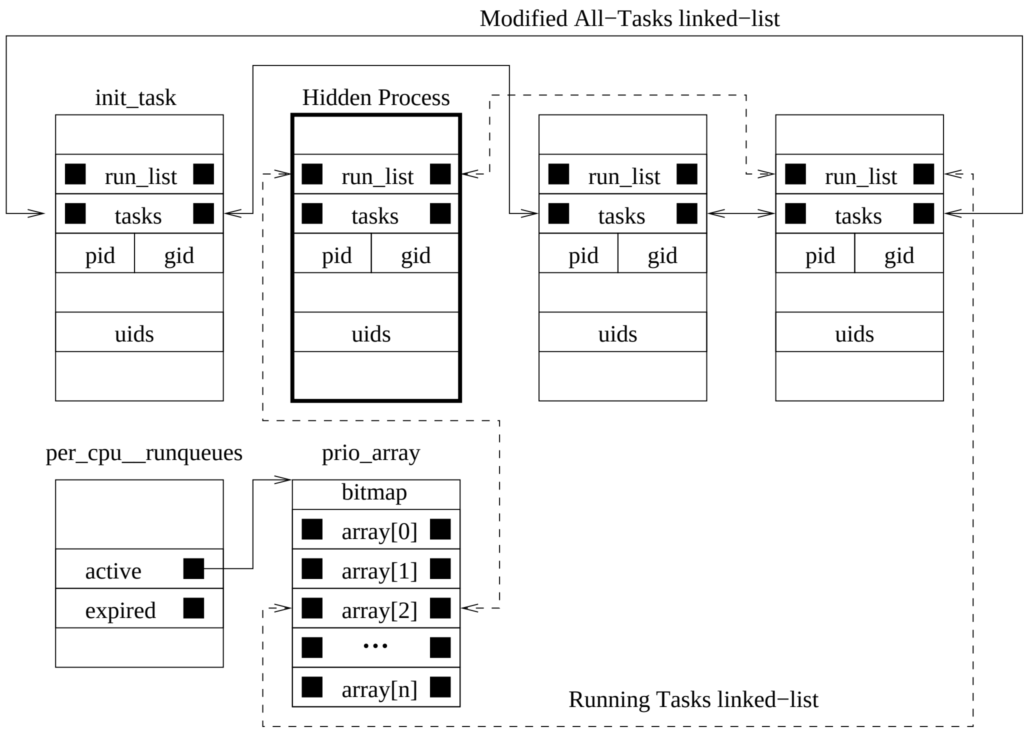

Figure 2.3: Data-only process hiding in Linux.

accounting. The dashed line shows the relationship between objects relevant to a particular processor's run queue, including tasks that are waiting to be run (or are currently running) on that processor. Even though the second depicted task is no longer present in the all-tasks list, it continues to be scheduled by the kernel. Two simple changes to dynamic data therefore result in perfect stealth for the attacker, without any modifications to control data, registers, or kernel text.

Chapter 3

From Threat Model to State-Based Monitoring

In this chapter, we draw upon our analysis of real-world kernel threats in the previous chapter to design a monitor capable of detecting runtime attacks against the operating system kernel. We begin by identifying the basic assumptions that frame the constraints for this work, based on our observations of attacker capabilities and trust relationships in modern computer systems. We then introduce a high-level model for the OS kernel and provide intuition for understanding kernel execution monitoring. Finally, we describe a specific type of execution monitoring, called state-based monitoring, and describe our high-level architecture for implementing such a monitor.

3.1 Threat Model and Assumptions

In order to design an effective kernel monitor, we must first understand the environment in which that monitor will operate. Based on our analysis of real-world kernel threats, we begin with a basic assumption about the attacker's capabilities and a corollary that follows from this assumption.

Assumption 3.1.1 (Attacker control) *We assume that an attacker has complete access to alter any software or storage on the protected system, including an ability to execute code at the processor's highest privilege level.*

Corollary 3.1.2 (Kernel untrustworthy) *The operating system kernel and, therefore, dependent userspace processes and libraries may not be trustworthy.*

Assumption 3.1.1 and its corresponding corollary are realistic for the vast majority of real-world environments. Particularly in light of the analyzed attacks described in the previous chapter, it is clear that attackers can and do find ways to compromise the kernel. As shown in Figure 3.1, modern computer systems can be viewed as a set of layers. Each layer relies on the integrity of all layers below it. Therefore, the compromise of any given layer reduces the dependability of all components at or above the compromised layer. Attackers frequently compromise systems due to flaws in one or more components. For example, a flaw in a network service may allow an attacker to gain access to a process executing as an unprivileged user. A second flaw, such as a user/kernel pointer vulnerability [54], may then be exploited to provide attackers with kernel-level access and, therefore, the ability to alter any software or storage on the system.

Similar access may be obtained when the credentials of trusted insiders become compromised — an equally plausible scenario in current deployments. Trusted insiders can load kernel extensions and disable security mechanisms, thereby enabling arbitrary system changes. Loadable kernel modules (LKMs), which are common in both Windows and Linux, have complete access to the kernel's address space and may be loaded in response to events not directly under a user's control. Buffer overruns and other vulnerabilities afford the attacker the ability to corrupt potentially arbitrary areas of kernel memory. Compromised hardware devices, and even stan-

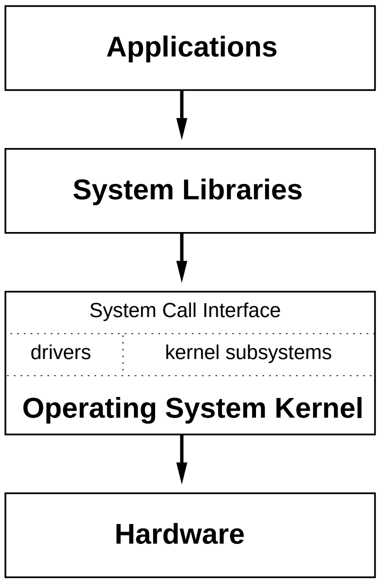

Figure 3.1: Dependencies in a typical modern computing environment.

dard virtual devices, such as Linux's `/dev/kmem` or portions of its `/proc` file system, may also provide the attacker access to kernel memory.

It directly follows from the above that we cannot build any solution that critically depends on, or operates as part of, the kernel itself. Accordingly, we must build upon other components of the system that will remain trustworthy. Specifically, we aim to remove the operating system kernel and all user applications from the *trusted computing base* (TCB) [73] of our kernel monitor. Instead, we utilize a much smaller set of dependencies, which we refer to as a *root of trust*.

Assumption 3.1.3 (Root of trust) *We assume that it is possible to develop a* root of trust *with access to kernel state that is not reliant upon the kernel's execution for access. Such a mechanism remains trustworthy and has a small amount of protected storage available to it. The attacker does not have the ability to directly* *control or modify this root of trust.*

Assumption 3.1.3 states that the only exception to attacker control in our target system is a small, trusted module capable of performing trusted execution. As we describe in more detail in Chapter 4, a variety of approaches exist for implementing this component. However, all current approaches rely on at least some portion of the machine's hardware remaining trustworthy.

Assumption 3.1.4 (Hardware trustworthy) *We assume the attacker does not have physical access to modify any hardware within the target system, and we further assume that this hardware operates as expected.*

As the lowest layer of the system, all software relies on the trustworthiness of hardware. Therefore, in nearly all systems, the integrity of the underlying hardware is treated as axiomatic. While mechanisms exist to verify hardware (and to prevent tampering), most organizations assume that the hardware they receive from a vendor is trustworthy. In most environments, the subsequent locking of computer cases and server rooms is sufficient protection. Laptops and handheld devices pose a more challenging problem for physical security, but anti-tamper mechanisms and vigilant users can reduce these risks.

It is worth noting that the line between hardware and software is sometimes blurry. For example, Intel microprocessors make use of updatable microcode extensions [47]. Additionally, most hardware devices, such as network cards and disk controllers, have some amount of mutable firmware that enable upgrades and fixes. It is therefore necessary in real systems to determine which components are essential for the particular root of trust in question. Further discussion of hardware security is beyond the scope of this work and we, like most users, assume that attackers do not have the capability to modify the physical plant of the machine.

3.2 Modeling and Monitoring the Kernel

Having described the problem of kernel-level integrity attacks and outlined the threat model that frames those attacks, we now turn our attention to the design of a monitor that is capable of detecting them. We begin by introducing a high-level model for the OS kernel and develop a pair of corresponding models for performing kernel monitoring.

The kernel, like any program, is composed of a set of processor instructions (referred to as *code* or *text*) along with data that can be operated on by those instructions. As the kernel executes, it transitions through a sequence of *states*, which describe the kernel's code and data at each point of execution, typically stored in the registers, RAM, caches, and on-disk (swapped) pages. Based on the sets of inputs, outputs, and possible states, we can characterize the behavior of a given kernel and monitor that kernel for unexpected behavior.

More formally, we can view an OS kernel K as an I/O automaton [65], whose elements are summarized in Table 3.1. An I/O automaton can be visualized as a labeled, directed graph, where nodes represent states $s \in \mathit{states}(K)$, and directed edges between s and s' with label π represent the transitions $(s, \pi, s') \in \mathit{steps}(K)$. Each label π identifies the action that accompanies the transition, and this action could be either input, output, or some internal event. Because states s include the kernel code as well as its data, this implies each state s contains an encoding of the transition relation $steps(K)$. (For simplicity, the formalism ignores the possibility of loading new code modules; modules the kernel might need are modeled by K.)

Actions	π	$\in acts(K)$
States	s	$\in states(K)$
Start states	$start(K)$	$\subseteq states(K)$
Transitions	$steps(K)$	$\subseteq$
$states(K) \times acts(K) \times states(K)$		

Table 3.1: I/O Automaton K (representing the kernel).

An attacker may wish to exploit a weakness in the hardware/software system of which K is a part to compromise K's integrity, effectively morphing K into some other automaton K' whose actions, states, and steps could all differ from those of K. The goal of the attacker is often to introduce an illicit capability into the system, and the kernel modifications either implement this capability or else hide its implementation in one or more user-level objects.

3.2.1 Runtime Kernel Monitoring

One way to defend against such attacks is to use an *integrity monitor*. The goal of an integrity monitor $M(K)$ is to detect when a kernel K has been modified. (We write M instead $M(K)$ when K is clear from context.)

A difficulty in designing an effective monitor is that attackers may compromise K by exploiting vulnerabilities in K itself. Ideally we would correct K to yield a kernel K_c that no longer contains the vulnerabilities, but this is difficult, and is indeed a motivation for monitoring in the first place. Likewise, devising a *perfect monitor*, by which an imperfect kernel is checked against a perfect specification, is equally impractical — if we could devise a perfect specification against which to check the kernel, we could instead fix the kernel.

Therefore, our goal is to devise a monitor that only checks a specific property or properties, rather than total correctness. We call such a monitor a *transition monitor* because it checks these properties at each transition.

Definition 3.2.1 (Transition Monitor) *A* transition monitor $M_{SP}(K)$ *of a kernel* K *accepts an approximation of* K*'s ideal kernel* K_c*: Given* $S \supseteq \mathit{states}(K_c)$ *and* $P \supseteq \mathit{steps}(K_c)$*,* $M_{SP}(K)$ *signals an integrity violation when it observes* K *enter a state* $s \notin S$*, or enter a legal state* s' *via an illegal transition* $(s'', \pi, s') \notin P$*.*

The property p that the monitor checks is defined by the choice of S and P, which are overapproximations of the ideal kernel ($S = \mathit{states}(K_c)$ and $P = \mathit{steps}(K_c)$ would be total correctness). We believe that the choice of property p should be based on the following two criteria.

Criterion 3.2.2 (Defeats likely attacks) *Since* S *and* P *are but an approximation of* K_c*,* M_{SP} *will fail to flag some violations, in particular those whose induced states fall within* $S \setminus \mathit{states}(K_c)$ *or whose transitions fall within* $P \setminus \mathit{steps}(K_c)$*. We wish to design* p *so that likely attacks fail to fall within these differences.*

Criterion 3.2.3 (Derived from the imperfect kernel) *Despite the fact that* K*'s implementation has flaws, e.g., vulnerabilities that enable attacks, it must be "correct enough" that we can devise useful (checkable) properties by examining it.*

3.2.2 State-Based Kernel Monitoring

State-based monitoring sacrifices some of the precision of transition monitoring for the sake of simplifying the implementation, reducing the programmer burden, and improving performance. The key idea is to monitor the system periodically, rather than after every transition. We express this idea precisely as follows.

Definition 3.2.4 (State-based Monitor) *A* state-based monitor $M_S^n(K)$ *periodically checks the legality of states, ignoring the transitions. More precisely: given a set* $S \supseteq states(K_c)$ *and an integer* n*,* M_S^n *signals a violation when, after* K *has taken* n *steps since it was last checked, it enters a state* $s \notin S$*.*

The benefit of a state-based monitor over a transition monitor is that it provides a tunable parameter for trading off performance with precision. Smaller values of n have greater precision, while larger values of n have better performance. Despite not dealing with transitions directly, state-based monitoring can be effective because the program's subsequent execution possibilities are captured by its current state, i.e., its code and data. Analyzing this state, the monitor can determine whether a property could be violated during later execution.

3.3 State-Based Monitor Architecture

Drawing on the system model and assumptions described previously in this chapter, we now introduce the architecture of our property-based kernel integrity monitor. Our approach utilizes a state-based monitor built on top of a root of trust,

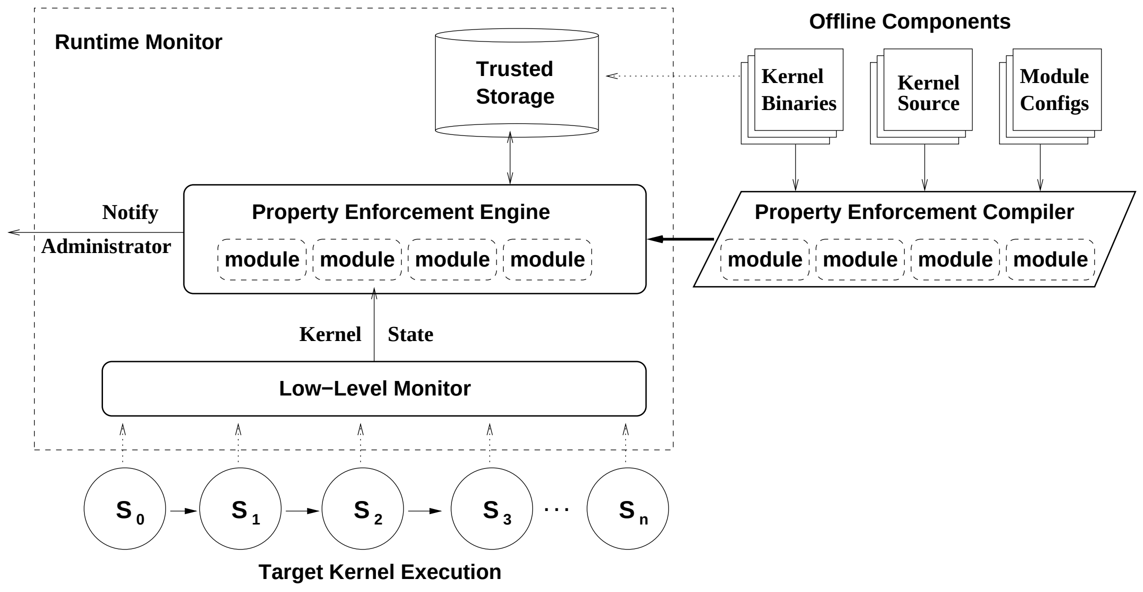

Figure 3.2: State-based kernel monitor architecture.

described in more detail in Chapter 4. We divide our system into a set of online (runtime) components and offline (pre-deployment) configuration components.

In order for the monitor to verify the running kernel's low-level state, it must be programmed with the appropriate property checking logic. There are many possible approaches for implementing this logic, particularly in light of the design considerations discussed in the previous section. The approach taken in this dissertation is to implement a modular system that allows new properties to be identified and multiple property verification procedures to be implemented, independent of the underlying monitor infrastructure. A set of common APIs and tools give module writers access to kernel state at various levels of abstraction and allow property verification module writers to extend these abstractions as necessary.

As shown in the right half of Figure 3.2, the primary offline component of our system is the property enforcement compiler, which takes as inputs the target kernel's source and binaries, a set of property verification modules, and module-specific configuration/specification data. The compiler implements a set of APIs that provide modules with access to various information about the kernel's source and binaries. Each module is responsible for implementing the verification of one or more properties for the target kernel.

Modules may also require additional configuration data, beyond information about the target kernel's source and binary. For example, one of our modules, discussed in Chapter 6, implements a policy language for describing properties of kernel data. The primary inputs to the module are therefore specification rules, which are then transformed by the module into low-level checks.

The output of the property enforcement compiler is a custom "enforcement engine" that verifies the implemented property checks for a specific kernel implementation and deployment. Our current implementation is a two-phase approach in which module verification logic uses automatic code generation to produce C code, which is then compiled and linked with the necessary low-level libraries for accessing kernel state.

The left half of Figure 3.2 depicts the online components of our system, which we now describe.

- *Low-Level Monitor.* As shown in Figure 3.2, the low-level monitor has access to the kernel's states ($s \in \mathit{states}(K)$, depicted as circles labeled S_1 through S_n), but not its transitions ($\mathit{steps}(K)$, depicted as arrows between the circles).

As described in Chapter 4, a number of mechanisms are available for implementing the low-level monitor, each with its own set of advantages and drawbacks. Because different mechanisms may be better-suited for different environments, we separate the low-level monitor from the business logic of state analysis and property verification by defining a uniform API for accessing the kernel's state via the low-level monitor.

- *Property Enforcement Engine.* The property enforcement engine represents the computational and policy assessment component of our runtime monitor. Based on the set of modules implemented in the offline stage, this component is responsible for determining whether the kernel's current state violates any of the implemented property checks — that is, whether the current state S_j is a member of the overapproximated set of states $S \supseteq \mathit{states}(K_c)$. If this check fails, then a monitored property has been violated and the administrator can be notified. As shown in the figure, certain property checks may require access to some amount of trusted storage that is accessible by the analysis engine at runtime. As just described, in our current implementation, the property enforcement engine is an automatically-generated program that is tailored to a particular kernel deployment.

In this dissertation, we present two property enforcement modules. First, in Chapter 5, we introduce an automated approach for performing state-based detection of persistent kernel control-flow modifications. In Chapter 6, we describe a specification language-based module that allows experts to easily specify additional high-level properties, which are automatically transformed into low-level checks.

When a property is violated, there is a security concern within the system. Depending on the nature of the violated property, the monitor may be configured to take actions varying from logging an error to notifying an administrator or even shutting down the system. While notification is the only response considered in this dissertation, other possibilities may be available. For example, it may be possible to perform some amount of automated repair or recovery, depending on the specific modifications that are detected. As demonstrated by Grizzard [37], Demsky and Rinard [25], and Bohra et al. [13], investigating other responses, such as direct repair, is a promising area of ongoing work.

Chapter 4

Low-Level Monitor Mechanisms

As described in the previous chapter, our runtime property-based monitor has two components: the low-level monitor, which is the mechanism for accessing the kernel's runtime state, and the property enforcement modules, which ensure that certain properties of this state hold true. In this chapter, we focus on the low-level monitor and present two proof-of-concept implementations.

4.1 Requirements and Design Goals

To perform its task of monitoring host state, an external monitor must meet, at a minimum, the following set of requirements. These requirements guide the implementation of a *root of trust* within the system, which is capable of monitoring the untrustworthy kernel without relying on it.

- *Unrestricted access.* The monitor must be able to access all of the kernel's state – its memory, registers, and any other storage, including on-disk pages (for kernels that page).

- *Sufficient resources.* The monitor must have sufficient resources to perform its duties of accessing and analyzing kernel state. Resource requirements, such as amounts of memory and processing, may depend on the specific property enforcement mechanisms chosen or other environmental factors.

- *Independence.* The monitor should not rely on the target kernel for access to resources, including main memory, logging, address translation, or any other task. The monitor must continue to function regardless of the running state of the target's software, particularly when the target has been compromised. One particular resource of interest is an *out-of-band reporting mechanism*. The monitor must be able to securely report the status of the host system. To do so, the monitor must not rely on the possibly-compromised host, e.g., for network or disk operations.

- *Assurance.* To the degree possible, we would like confidence that the monitor is implemented such that it achieves the above goals. Ideally, the monitor will be simple and small enough that it can be sufficiently verified.

4.2 Copilot: A Coprocessor-Based Integrity Monitor

Our first implemented monitor mechanism is a coprocessor-based monitor, called Copilot, that is capable of accessing host memory without kernel intervention.[1] In its current form, Copilot is implemented as a custom add-in board that operates from the Peripheral Component Interconnect (PCI) bus, found in the majority of current x86 computers. Figure 4.1 shows how Copilot can be used in a target machine. While the specifics of each chipset vary, the figure represents a typical configuration for a commodity system [38].

As shown in Table 4.1, Copilot is a fully independent hardware solution, based

[1]The Copilot monitor mechanism was jointly developed with the help of Tim Fraser, Jesus Molina, and Bill Arbaugh [83].

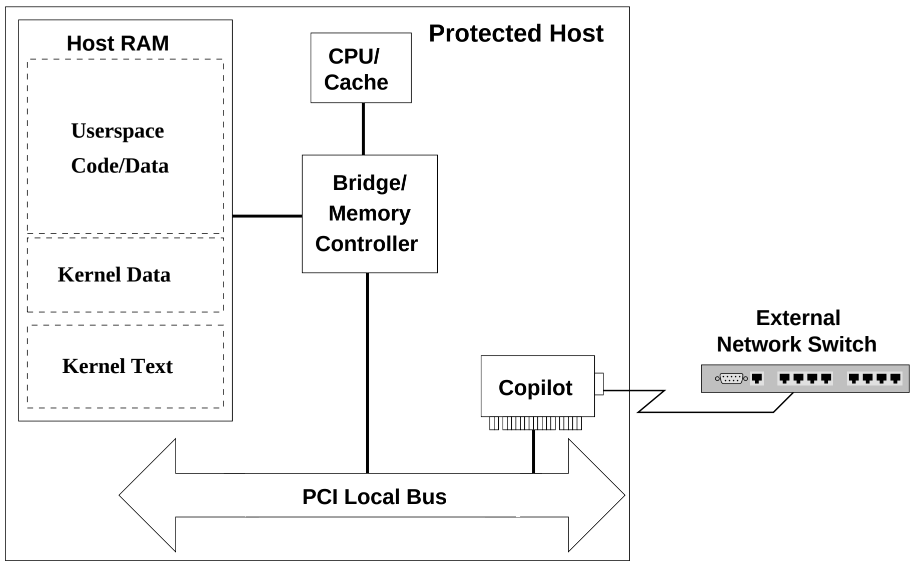

Figure 4.1: Copilot PCI coprocessor-based monitor.

on the IBM PowerPC 405GP processor [46]. In addition to its powerful processor, Copilot includes its own RAM, local storage, network card, and even an external power supply that allows it to operate when the host machine is powered down. The PowerPC 405GP board has full bus master functionality (described shortly), which allows for direct access to the host's memory [46]. Our current implementation utilizes a minimal embedded GNU/Linux distribution, which includes a custom kernel module and userspace library. The kernel module and library facilitate complete host memory access for the enforcement engine, which runs as a userspace process.

4.2.1 Memory Access

The PCI bus was originally designed for connecting devices to a computer system in such a way that they could easily communicate with each other and with the host processor. As the complexity and performance of these devices increased,

External Interface:	32-bit 66MHz PCI add-in (Spec. 2.2)
	Bus-mastering
Processor:	IBM PowerPC 405GP RISC
Persistent Storage:	4MB Flash memory
RAM:	32MB
Networking:	10/100 Ethernet on-board NIC
Power:	PCI bus or external A/C interface
Operating System	Embedded GNU/Linux 2.4 kernel

Table 4.1: Copilot board platform summary.

the need for direct access to system memory without processor intervention became apparent [81]. The solution provided by manufacturers has been to separate memory access from the processor itself and introduce a memory controller to mediate between the processor and the many devices that request memory bandwidth via the bus. This process is commonly referred to as direct memory access (DMA) and is the foundation for many high-performance devices found in everyday systems [81].

On any given PCI bus, there are two types of devices: initiators, or bus masters, and targets [95]. As the names suggest, bus masters are responsible for starting each transaction, and targets serve as the receiving end of the conversation. A target is never given access to the bus without being explicitly queried by a bus master. For this reason, bus mastering is a requirement for a device to utilize DMA. Most modern PC motherboards can support multiple (five to seven is typical) bus masters at any one time, and all reasonably performing network, disk, and video controllers support both bus mastering and DMA.

Since the ultimate goal of DMA is to facilitate resource sharing between a device and the processor, some information must be communicated by both parties to determine which portion of memory should be used. To account for addressing differences between the bus and main memory, the host operating system will typically calculate a translation from the host's memory address space to the PCI bus's address space and directly notify the device where in the latter it should access [81].

Unfortunately for a monitor that operates based on host addresses, this separation makes it difficult for a device to independently determine where in main memory it is actually reading or writing; there is no requirement that an easy reverse mapping from PCI bus addresses to system memory addresses exist. However, in the case of the PC architecture, the designers have set up a simple one-to-one mapping between the two address spaces. Therefore, any access to PCI memory corresponds directly to an access in the 32-bit physical address space of the host processor. The result is full access to the host's physical memory, *without* intervention or translation by the host's operating system.

To understand the steps involved when Copilot requires access to kernel memory, we now walk through a simplified example. In this example, the enforcement engine, running as a userspace process on the coprocessor, needs access to a critical section of the host kernel's memory. Figure 4.2 depicts a number of *address spaces* present in either the host system, on the left, or the Copilot monitor, on the right. Each device contains one or more virtual address spaces, which are each mapped through the respective processor's virtual memory system to some amount of physical memory. As just described, the host platform's physical memory is linearly mapped directly into the PCI bus address space by a bridge on the host's motherboard. As a fully-functional on-board computer, the coprocessor also has its own peripherals and a communication bus (labeled "local bus") that links these devices.

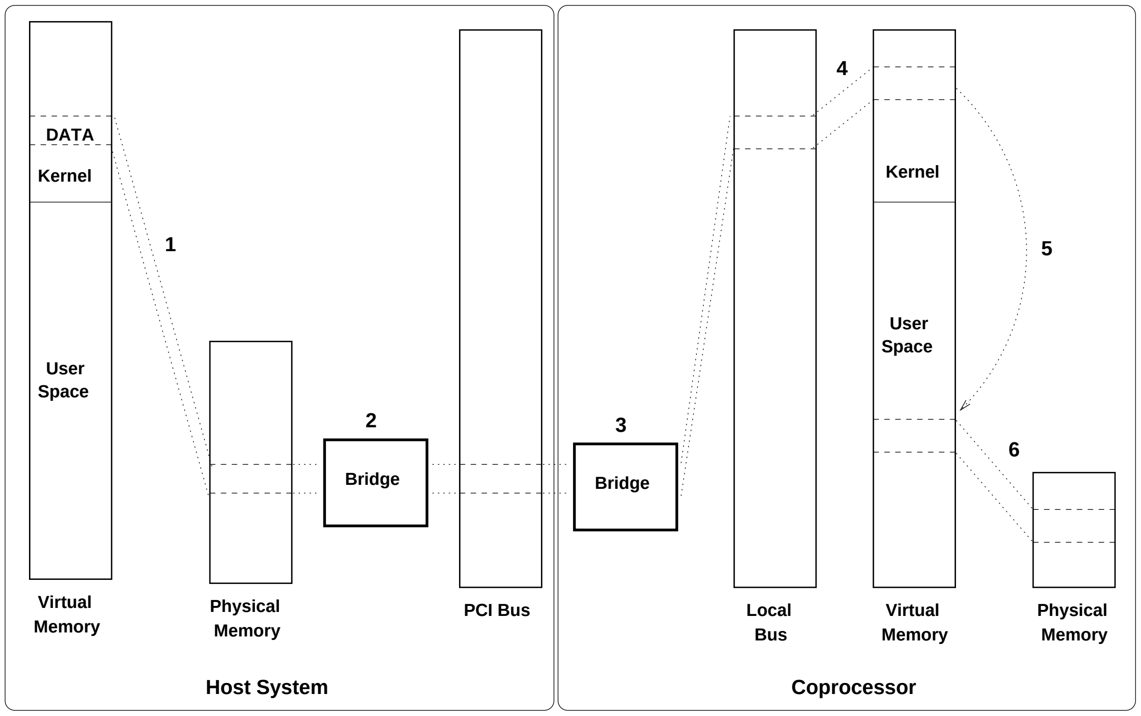

Figure 4.2: Host memory access from Copilot coprocessor.

The local bus is connected to the host's PCI bus via an on-chip PCI-to-PCI bridge.

Moving from left to right, we now explain the steps by which the enforcement engine receives the expected data. Because the data is identified by its host kernel virtual address, but the coprocessor only has direct knowledge of the host's physical addresses, Copilot must first emulate the host processor's virtual address translation (step 1) to locate the data within physical memory. This translation proceeds by referencing the same in-memory page tables that the host processor's memory management unit (MMU) uses to calculate physical addresses from virtual addresses (referred to as "linear addresses" by Intel [47]). Using the calculated host physical address, the coprocessor makes a request for the target data, which travels through the host's memory controller and bridge (step 2), across the PCI bus, through the PCI-to-PCI bridge (step 3), and onto the coprocessor's local bus. A portion of the local bus is mapped to a section of the Copilot Linux kernel's virtual memory address space using memory-mapped I/O (step 4) [14]. When the requested data arrives, it is copied to a userspace buffer in the enforcement engine's process space (step 5), the address of which is automatically translated to the coprocessor's physical memory address space by the hardware MMU (step 6).

4.2.2 Limitations

While Copilot is extremely effective at accessing kernel memory for most systems, there are a few drawbacks to a coprocessor-based approach. First, Copilot does not have access to the host's processor registers. This is particularly crippling because of the required address translation emulation described in the previous section. While most kernels maintain the necessary page table information at well-known physical addresses, simple changes to the processor's address translation registers can easily circumvent this dependency. One possible solution to this problem would be to combine our PCI monitor with a limited SMM monitor, described in Chapter 7, that simply verifies or communicates critical processor registers.

Additional challenges facing Copilot stem from its placement on the PCI bus, far from the running processor. While this is an advantage in terms of isolation and portability, this separation comes at a price. A creative attacker may find ways to disable the device – the most notable of which is sending a PCI-reset that will cause the bus to reboot. However, two caveats to this attack are worth noting. First, there is no easy interface for sending a PCI reset in most systems without rebooting the machine. Rebooting the host machine serves to bring unnecessary attention and may not be advantageous to the attacker. Second, because Copilot has its own power source and communication channel, the device can simply generate an alert when it detects that the host is no longer accessible.

More recently, hardware changes to support new bus architectures and virtualization have introduced additional mechanisms that can be used by an attacker to evade coprocessor-based monitors. As described by Rutkowska, one such attack is possible because of hardware devices that allow host physical memory addresses to be remapped to the bus address space (e.g., for memory-mapped I/O) [88]. On platforms equipped with this hardware, attackers can cause ranges of physical memory to be spoofed from the point of view of a device, but not the host processor. This decoupling causes the monitor to wrongly process what it believes to be kernel state, which could potentially lead to false negatives. More advanced forms of this attack will become trivial as hardware extensions to support virtualization, such as Intel's VT-d [50] and AMD's IOMMU [4] become widespread. As is the case with limited register access, these redirection attacks can easily be addressed through the implementation of a hybrid monitor that uses both software (such as SMM or VMM) and the hardware coprocessor. In such a system, a small piece of trusted code running in SMM or a hypervisor could protect and verify the coprocessor's access to a particular region of memory.

Fortunately, the same hardware features that create challenges for PCI-based monitors drastically improve the viability of virtual machine-based monitors.

4.3 Xen Virtual Machine-Based Integrity Monitor

Our second implemented low-level monitor mechanism is capable of analyzing virtualized guest operating systems running on top of the Xen Open Source hypervisor [10]. Unlike Copilot, our VMM-based monitor does not require extra hardware and has full access to all of the target virtual machine's state, including registers.

The term *virtualization* has been applied in many contexts to a number of different technologies and techniques. Generally speaking, the term refers to methods and mechanisms for partitioning and sharing computing resources. One approach to virtualizing resources is to provide a common machine or hardware-level abstraction that is available to one or more *virtual machines*, which utilize and manage resources provided by the abstracted layer. While the particular abstraction that is presented can vary dramatically, a common approach is to enable the same (or similar) view of system resources that real hardware would have provided.

In most implementations, a software layer called a *virtual machine monitor* (also called a *hypervisor*) is responsible for providing the machine abstraction and for managing and allocating the real underlying resources. There are many possible architectures for VMM design, but two common approaches are to replace the operating system entirely with a VMM layer (sometimes referred to as a Type I VMM) or to implement the VMM on top of an existing operating system, such as in a driver or process (Type II VMM) [87].

In this work, we focus on virtualization technologies that allow unmodified operating systems to run efficiently on commodity hardware. One challenge to these systems is that, until recently, x86 processors did not support hardware-level virtualization due to a number of so-called “sensitive but unprivileged” processor instructions that make it difficult to run commodity operating systems at lower privilege levels (called “rings” on x86) [87]. In the late 1990s, VMware Inc. developed techniques to work around this limitation through the use of intelligent binary rewriting for sensitive instructions [103]. More recently, both Intel and AMD have added a collection of instruction set extensions that allow the processor to be effectively virtualized. These are referred to as VT [48] and SVM [5] respectively.

Xen [10] is an Open Source VMM that benefits from a strong development community, including a large amount of commercial support from software and hardware vendors. The primary insight of the Xen approach is that, by presenting a more general machine abstraction to guest operating systems and porting each OS to that abstraction, a far more efficient sharing of resources is achieved. This approach is referred to by the Xen inventors as *paravirtualization*. However, Xen also supports unmodified guest kernels on platforms enabled with Intel’s VT or AMD’s SVM. As shown in Figure 4.3, Xen’s architecture relies on a very small hypervisor layer that multiplexes processor and memory resources and provides a common hardware abstraction to one or more guests, which are partitioned into isolated “domains.” One privileged domain, called domain 0, is used to facilitate I/O operations, as well as virtual machine management functions such as setup, save, and restore.

We have chosen the Xen hypervisor as the platform for our proof-of-concept VMM-based monitor for a number of reasons. First, as an Open Source project, we

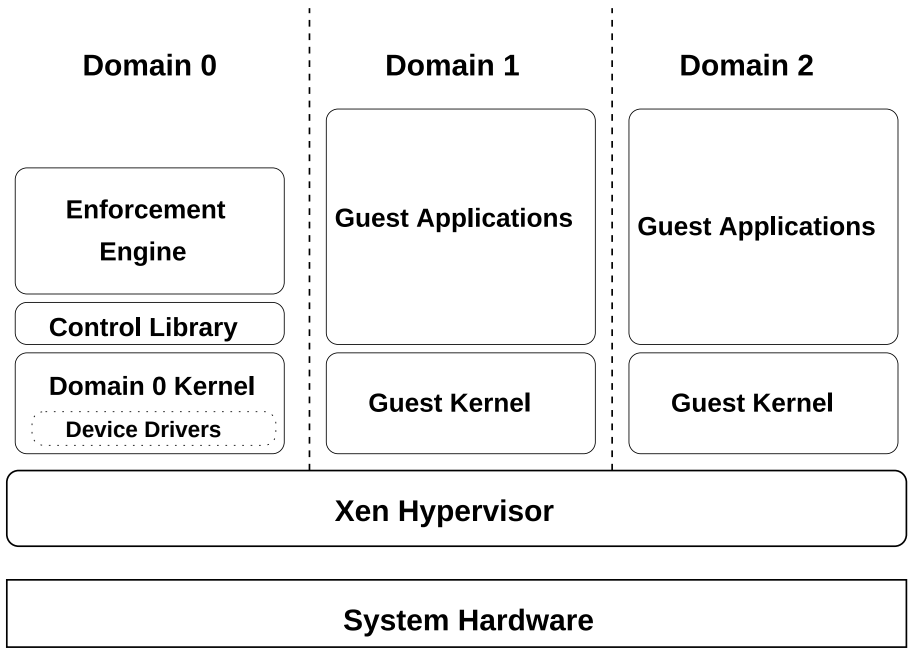

Figure 4.3: Xen VMM-based runtime monitor setup.

have complete access to understand and modify any of Xen's functionality. Second, the commercial support provided to Xen makes it a practical platform for real-world deployments. Next, Xen's feature set aligns closely with the requirements of this work – the lightweight hypervisor, support for unmodified guest kernels, and integrated mandatory access control system make Xen a promising choice for building secure, practical systems. Additionally, support for new processor features is constantly being added to the project. Finally, we found that Xen's architecture lends itself to easily implementing a low-level monitor, as we describe next.

4.3.1 Guest State Access

Because domain 0 is responsible for critical management tasks, such as loading virtual machines at boot time and saving their state at shutdown, the control interface exported to domain 0 by the hypervisor gives it a powerful feature set for runtime monitoring. Specifically, two types of requests, or *hypercalls*, enable the straightforward implementation of guest state analysis – memory mapping and processor state access.

As shown in Figure 4.3, our Xen-based property enforcement engine runs as a process within domain 0. To obtain access to a target domain's physical memory, the enforcement engine executes a memory-mapping hypercall (through a library interface) that maps a subset of the target's physical memory into the virtual address space of the enforcement engine process. Because of Xen's abstraction on machine memory (i.e., the actual hardware RAM present in the system) called pseudo-physical memory, one level of indirection is necessary to identify the machine pages that must be mapped. As with Copilot, access to physical memory means that the virtual address translation of the target machine must be emulated (alternatively, the hypervisor could be used as a translation oracle). However, unlike Copilot, our Xen-based monitor also has access to processor registers and can therefore guarantee that the correct page tables are in use.

Access to processor registers is also enabled via the hypercall interface. A memory buffer, passed as a parameter to a hypercall, is used to copy the current values of all processor registers for the requested domain's virtual CPU(s) from the hypervisor to the requesting process. Because the existing Xen interface provides only a subset of registers, we implemented a small patch (about 10 lines) to augment the register set. This patch represents the only change we required to enable full monitoring on the Xen platform.

4.4 Future Work

While our current low-level monitor implementations are both effective and practical, there are several possible improvements that could be made. First, it may be possible to improve the performance and effectiveness of the VMM-based monitor by more tightly integrating it with the hypervisor itself. For example, the hypervisor could be modified to directly enforce a more strict environment, thereby reducing the set of checks required by the monitor. Additionally, the monitor could be modified to be more "event driven," checking only when certain key guest VM events occur, as signaled by the hypervisor.

Second, as support for new trusted computing hardware becomes more widespread, the VMM-based monitor could be moved to an isolated partition outside of domain 0 and protected using attestation and late launch technologies, which are described in more detail in Chapter 7. Loscocco et al. have independently developed a Xen-based system similar to ours and propose these added protection measures [63].

Third, our current Xen implementation supports only unmodified guest kernels. It would be trivial to extend our technique to monitor paravirtualized guests, as demonstrated by the XenAccess project [82].

Finally, a hybrid VMM/coprocessor system has the potential to strike an appealing balance between the advantages of each approach. When combined with the control and visibility of a hypervisor (or SMM-based monitor, described in Chapter 7), the performance and isolation benefits of a coprocessor make it an excellent choice for offloading valuable cycles from application CPUs and network cards.

Chapter 5

State-Based Control-Flow Integrity

Having described the low-level mechanisms by which our property-based monitor gains secure access to the kernel's state, we now turn our attention to the details of property enforcement — that is, which properties of the kernel's state should be checked and how to effectively implement those checks. This chapter presents the first of two property modules that we have developed. This module enforces a property that we call *state-based control-flow integrity* (SBCFI).[1] A violation of SBCFI suggests that an attacker has introduced illicit functionality into the kernel by making a persistent modification to its normal execution. We begin our discussion by describing *control-flow integrity* (CFI), a stronger property on which SBCFI is based, and argue that violations of CFI are highly-correlated with an attacker's goals. As we explain, complete CFI monitoring is impractical for OS kernels. Therefore, we propose SBCFI as an alternative and present the details of its implementation.

5.1 Control-Flow Integrity

A program P satisfies the CFI property so long as its execution only follows paths according to a *control-flow graph* (CFG), determined in advance. If this graph approximates the control flow of the unmodified P, then a violation of CFI signals

[1]SBCFI was developed jointly with Mike Hicks [85].

that P's integrity has been violated. CFI enforcement has been shown to be effective against a wide range of common attacks on user programs, including stack-based buffer-overflow attacks, heap-based "jump-to-libc" attacks, and others [2].

More generally, the goals of rootkits are squarely at odds with CFI. Since the attacker's main goal is to add surreptitious functionality to the system, then either this functionality or the means to hide it are most easily enabled by modifying the kernel's CFG to call injected code. Our analysis of Linux rootkits described in Chapter 2 shows that an overwhelming majority of them, 24 out of 25 (96%), violate the control-flow integrity of the kernel in some way. As far as we are aware, we are the first to make this observation. Additionally, our preliminary analysis of about a dozen Windows kernel rootkits demonstrates a similar trend among those threats. This suggests that CFI is a useful property to monitor in the kernel.

Abadi et al. [2] enforce CFI for a program P by rewriting P's binary. The target of each non-static branch is given a tag, and each branch instruction is prepended with a check that the target's tag is in accord with the CFG. This strategy provides protection against an attacker with access to P's memory under three conditions: (1) tags must not occur anywhere else in P's code; (2) P's code must be read-only; and (3) P's data must be non-executable. These assumptions are easily discharged for applications. The first assumption is discharged by rewriting the entire application at once, preventing conflicts, and the latter two are discharged by setting page table entries and segmentation descriptors appropriately; as the page tables can only be modified within the kernel, it is assumed they are inaccessible to the attacker.

Unfortunately, these assumptions cannot be discharged as easily when monitoring the kernel itself. An attacker with access to kernel memory could overwrite page table entries to make code writable or data executable, violating assumptions (2) and (3). It is also unrealistic to expect to rewrite all core kernel code and LKMs at the outset, and thus it is difficult to discharge the first assumption and avoid tag conflicts. Moreover, it is nontrivial to compute a precise CFG for the kernel in advance, due to its rich control structure, with several levels of interrupt handling and concurrency.

5.2 State-Based Control-Flow Integrity

We propose to enforce an approximation of CFI using a *state-based monitor* (Chapter 3); we call the resulting property *state-based control-flow integrity* (SBCFI).

We enforce SBCFI by ensuring that the CFG induced by the current state is not different from the CFG of the original deployed kernel. The details of our implementation are presented in Section 5.3. In summary, our approach has two steps:

1. *Validate kernel text, which includes all static control-flow transfers.* The monitor keeps a copy or hash of K's code. At each check, it makes sure K's code has not been modified by comparing it against the copy or hash. This ensures that static branches occurring within the kernel (e.g., direct function calls) adhere to the kernel's CFG.

2. *Validate dynamic control-flow transfers.* To validate dynamically-computed branches, the monitor must consider the dynamic state of the kernel — the heap, stack, and registers — to determine potential branch targets. Our implementation relegates its attention to function pointers within the kernel. Analogous to a garbage collector, the monitor traverses the heap starting at a set of roots — in our case, global variables — and then chases pointers to locate each function pointer that might be invoked in the future. It then verifies that these pointers target valid code, according to the CFG.

Because it monitors the kernel's state periodically, an SBCFI monitor can only be used to reliably discover *persistent* changes to the kernel's CFG: if an attacker modifies the kernel for a short period, but undoes his or her modifications in time shorter than the monitoring period, then the monitor may fail to discover the change.

Nevertheless, limiting our attention to persistent modifications to the CFG is extremely useful. Consider Table 2.1 again. Recall that for each modification that we identified in columns 7–10, we also determined whether the modification is persistent, represented by a P, or transient, represented by a T. Columns with a B indicate that both persistent and transient modifications of a given type are made. We found that all of the analyzed threats make some persistent change to the kernel. Furthermore, in all of the attacks in which a control-flow modification is made, some portion of those changes was found to be persistent. Thus the same 24 rootkits that violate CFI also violate SBCFI.

This makes sense from the attacker's perspective: the goal of a rootkit is to introduce surreptitious, long-term functionality into the target system, e.g., to facilitate later reentry, reconnaissance (keystroke monitoring, packet sniffing, etc.), or defense neutralization. Changes to the kernel CFG to facilitate this are thus *indefinite*, so SBCFI will discover them, even for large monitor intervals.

As an example of how SBCFI can be used to detect threats that persistently modify the control-flow of the kernel, consider the `Linspy` Linux rootkit. `Linspy` is a kernel-level keystroke logger that modifies the kernel in two ways. First, it redirects the `write()` system call to look for keystroke events from file descriptors that are associated with Linux terminals. When keystroke data arrives for a device of interest, copies of that data are placed in a kernel buffer before the kernel's real `sys_write()` is called. Second, `Linspy` registers a new character device, e.g., `/dev/linspy`, to provide a malicious userspace process with access to the collected data. This step is performed by utilizing the VFS's built-in character device infrastructure through a call to the `register_chrdev()` function. One of this function's parameters is a structure containing a set of implemented function pointers, such as the `readdir()` callback described in Section 2.2.1. `Linspy` results in five SBCFI violations – one for the modified system call, and four for the registered callback functions (`open()`, `release()`, `read()`, `ioctl()`) associated with the added character device.

The only attack that we encountered that does not cause a persistent control-flow modification is the `hp` rootkit. `hp` performs simple process hiding by removing the process from the kernel's "all tasks" list while leaving it in its "to-be-scheduled tasks" list, as described in Section 2.2.3. Other attacks utilize a similar technique for hiding modules, but must make control-flow modifications elsewhere to enable execution of the module's code.

While an SBCFI-based monitor detects many attacks that would not be detected by previous kernel integrity monitors, it is not a panacea. Current attacks always inject some persistent modification, but an attacker could avoid detection by persistently applying transient attacks. For example, a remote host could regularly send a packet that overruns a buffer to inject some code which gathers local information and then removes itself, all within the detection interval. Even so, this form of attack limits what the attacker can do compared to having persistent code in the kernel itself, e.g., to log keystrokes. A clever attacker might find a way to corrupt a kernel data structure so that periodic processing in the kernel itself precipitates the buffer overrun. Though much harder to construct, such an attack would help avoid detection via a network intrusion detection system, and could make information gathering more reliable.

Our implementation of SBCFI is also limited because we relegate our attention to function pointers and not other forms of computed branch, such as return addresses on the stack, or in the extreme case, function pointers manufactured by some complex computation. Not considering return addresses prevents detection of stack smashing attacks; however, since these attacks are typically transient, a state-based monitor is unlikely to detect them anyway. Additionally, our monitor could miss modifications to portions of the stack that are long-lived. To our knowledge, function pointers in the kernel are usually stored in record fields directly, and not computed.

In short, though SBCFI may miss attacks that would be detected by CFI, it is straightforward to build a SBCFI monitor that is protected from tampering; that can detect the types of mechanisms used by all of the control-flow modifying rootkits that we could find; and that significantly "raises the bar" for constructing new attacks.

5.3 Implementation

To evaluate the usefulness of our approach, we implemented a state-based CFI monitor for the Linux kernel as a module in our property verification infrastructure and tested our implementation using our Xen-based VMM monitor.

Much of the monitor process is generated automatically from the target kernel's source code and compiled binary, as shown in Figure 5.1. The generation proceeds in several stages. The *Type & Global Extractor*, *Symbol Mapper*, and *Type Mapper* are used to gather information about the target kernel's symbols and type structure. This information is passed as configuration input to the *SBCFI Monitor Generation Module*, a Python module for our property enforcement compiler that generates C code to traverse the kernel's heap to look for function pointers. Along with code from any other included modules, the generated code is linked against VMM-specific routines in the monitor library for accessing the target's memory.

As described in Section 5.2, to verify that the kernel's control-flow has not been modified, the kernel monitor performs two tasks: (1) it validates that the kernel's text has not been modified and (2) it verifies that all reachable function pointers are in accord with the kernel's CFG. We discuss each of these points in turn.

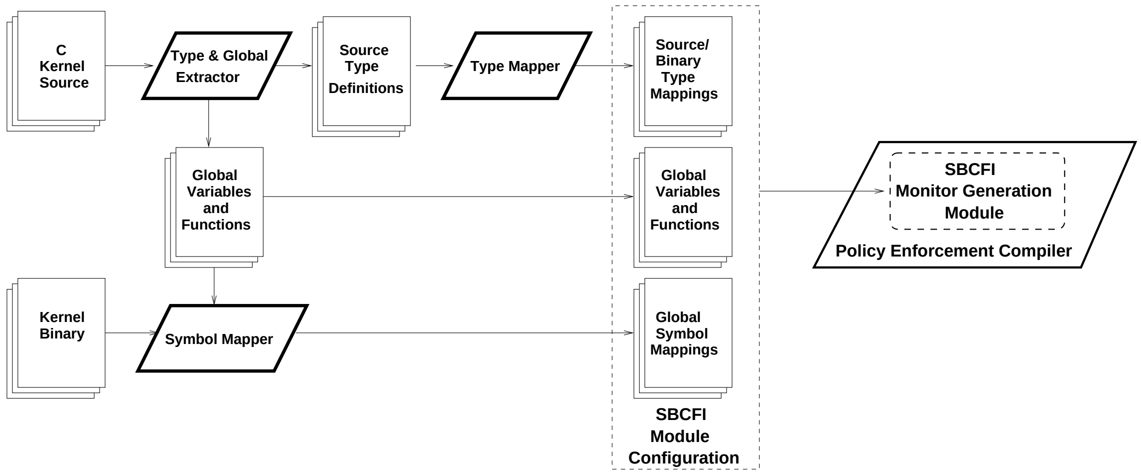
C Kernel Source
Type & Global Extractor
Source Type Definitions
Type Mapper
Source/ Binary Type Mappings
Global Variables and Functions
Global Variables and Functions
SBCFI Monitor Generation Module
Policy Enforcement Compiler
Kernel Binary
Symbol Mapper
Global Symbol Mappings
SBCFI Module Configuration

Figure 5.1: Monitor generation process and components.

5.3.1 Validating Kernel Text

For the Linux kernel, the set of allowable runtime code is determined by two sources: (1) the static portion of the kernel that is loaded by the boot loader and (2) a set of authorized loadable kernel modules (LKMs), which can be loaded or unloaded dynamically during kernel execution. The generated monitor takes as input trusted copies of the kernel and LKM binaries for runtime comparison (this is shown by the dashed line in Figure 5.1). The code verification procedure works as follows:

1. Compare the executable sections of the static kernel in the trusted store with those in memory. If equal, add the sections to the set of verified code regions V; otherwise, add them to the invalid code regions set I.

2. Traverse the list of kernel LKMs kept by the target kernel[2] to locate all cur-

[2]The address of the root of this list is determined by examining the trusted static kernel binary.

rently loaded modules. For each kernel module:

   (a) Locate the trusted copy of the LKM. If no trusted copy can be located, add the module to I.

   (b) Emulate the module loader to adjust all relocatable symbols in the trusted LKM copy based on where the module is loaded in target memory.

   (c) Compare the text sections of the emulated copy to what is in memory at the expected location. If equal, add the sections to V; otherwise add them to I.

3. Report the set of invalid code regions I if nonempty. (The set V is used in the next phase.)

Step 2b is necessary because LKMs are relocatable. When emulating dynamic linking, a module's external references to kernel symbols are resolved to what the monitor believes is potentially valid code and data; i.e., the targets must reside in the text or static data area of the core kernel or one of the modules in the module list. This avoids having to trust the kernel's module symbol table and has the effect of validating any static, inter-module function calls.

To make text verification more efficient, we applied two optimizations to this algorithm. First, we use a cryptographically secure hash algorithm to speed up all comparisons (Steps 1 and 2c). Second, we cache the hashes of the relocated, trusted LKMs computed in Step 2b. We can reuse these when comparing to in-kernel modules whose position has not changed since the last check.

5.3.2 Validating Dynamic Control-flow

Validating the kernel's text ensures that all static control-flow transfers — in particular, direct function calls — are in accord with the kernel's CFG. The monitor must also validate all dynamic control-flow transfers; i.e., those for which the transfer target is not known until run-time. For the x86 architecture, the two main sources of dynamic transfers are (1) indirect calls to functions (i.e., via function pointers) or labels (e.g., as part of a `switch` statement) and (2) function call returns (regardless of whether the function was called directly or indirectly). The latter category is typically implemented by popping a return address off of the stack and jumping to it, e.g., via the `return` instruction.

As already discussed, our kernel monitor does not consider function call return targets or intraprocedural dynamic branch targets. This is because such attacks, in and of themselves, do not create a persistent integrity violation in the kernel. This leaves us with the task of verifying the targets of function pointers that might be used by the kernel during later execution. The first step is to identify the set of possible function pointers. A reasonable approximation of this set is those function pointers *reachable* from a set of roots via a chain of pointer dereferences, in the spirit of a garbage collector (GC). We can construct a traversal algorithm to find the reachable function pointers, given three inputs: (1) the set of initial roots (e.g., global variable addresses, the stack, and the registers); (2) the offsets within each object at which there are pointers; and (3) an indication of which pointers within an object are function pointers. With this, a traversal algorithm can start at the roots and transitively follow the pointers embedded in objects it reaches until all function pointers have been discovered.

We gather the necessary inputs via static analysis of the kernel's source code and compiled binary, and the monitor generator module constructs the traversal code in three steps:

1. From the kernel source, extract all global variables and their types (Section 5.3.2.1).

2. Construct a *type graph* based on the type definitions occurring in the kernel source. Each node in the graph represents a type, and an edge from T_1 to T_2 implies objects of type T_1 contain a pointer (or pointers) to objects of type T_2. The graph includes only types from which function pointers can ultimately be reached (Section 5.3.2.2).

3. Using the global variables as starting points and the type graph as a specification, generate code to locate all function pointers reachable from global variables (Section 5.3.2.3).

As it discovers reachable function pointers, the traversal algorithm will then validate those pointers according to an approximation of the CFG, described in Section 5.3.2.4.

5.3.2.1 Finding the Roots

The first step in generating the traversal code is to identify the roots, which include the program's global variables, the stack, and the registers. The *Type &*

Global Extractor (Figure 5.1) extracts the global variables and their types from the kernel source using a simple C Intermediate Language (CIL) [74] module.

We do not consider the stack and most of the registers in our traversal code because, unlike global variables, their contents cannot be given a static type: they will contain values of different types depending on the current program counter and calling context. To address this issue in a garbage collection setting, the compiler can generate metadata used by the GC traversal to designate the types of the registers and stack frames at various program points. Constructing such a compiler for C would be a substantial undertaking, and would be complicated by C's weak type system (discussed below). In the absence of such data, a conservative garbage collector [12] can pessimistically regard any stack or register word as a pointer if it falls within the range of legal memory (among other validation criteria). In our setting, this approach is insufficient as we also need to know the *type* of that memory, in order to know whether it contains function pointers that we must validate. We consider the x86 registers `idtr`, `gdtr`, `sysenter`, the debug registers, `eip`, and the `cr` registers as roots because the type of their contents is fixed.

Because we do not consider the complete root set, we will miss some CFG modifications; for example, we will not notice modified code pointers on the stack or in untyped registers, nor will we notice modified code pointers in objects reachable only from these locations. Nevertheless, because the contents of the stack and the untyped registers are transient, ignoring them should not cause us to miss persistent attacker modifications.

5.3.2.2 Constructing the Type Graph

The next step is to construct the type graph. This happens in two steps. First, the *Type & Global Extractor* extracts all type definitions from the kernel source. With these as input, the *Monitor Generator* builds the type graph $G(n, e)$ using the procedure in Figure 5.2. Figure 5.3 depicts the type graph for the (simplified) types found in Figure 5.4, with the function-pointer containing structures highlighted (`dentry_operations`, `inode_operations`, `super_operations`, and `file_operations`; definitions not shown in Figure 5.4).

```
procedure BUILDTYPEGRAPH()
  $Nodes \leftarrow$ set of extracted types from kernel source
  $Edges \leftarrow \emptyset$
  $FPNodes \leftarrow \emptyset$
  for each Struct $s \in Nodes$
    do for each member $m \in Members(s)$
         do if $IsFunctionPointer(m)$
              then $FPNodes \leftarrow FPNodes \bigcup \{s\}$
            if $IsStruct(m)$
              then $Edges \leftarrow Edges \bigcup (s, Type(m))$
  for each Struct $n \in Nodes$
    do $ExcludeNode \leftarrow$ true
       for each Struct $f \in FPNodes$
         do if $PathExists(Edges, n, f)$
              then $ExcludeNode \leftarrow$ false
                   break
       if $ExcludeNode$
         then $RemoveNode(Nodes, Edges, n)$
  return $(Nodes, Edges)$
```

Figure 5.2: Algorithm to generate type graph.

Unfortunately, the type information in the kernel source is not sufficient to identify all typed pointers. Because C's type system is not expressive enough to

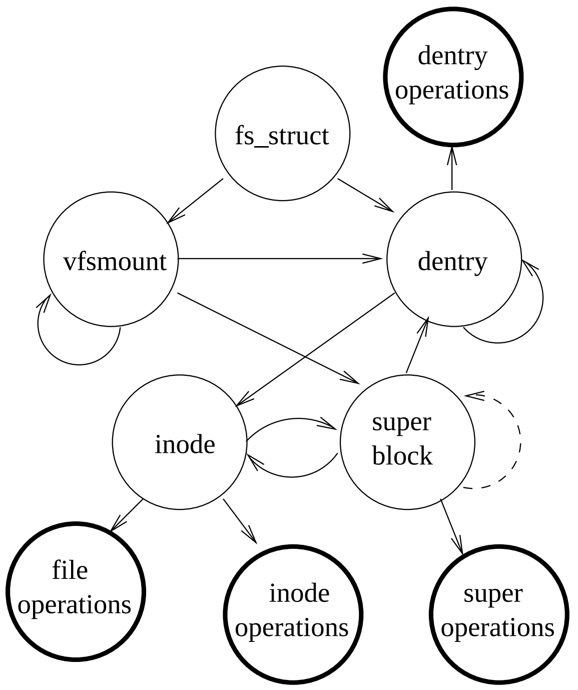

Figure 5.3: Function pointer reachability graph.

specify some useful idioms, programmers use conventions to encode them. For example, C does not provide polymorphism (generics), so programmers often cast generic elements to/from `void*` (or even `int`). Similarly, the Linux kernel makes heavy use of list data structures embedded in other objects, and the precise type of the target object of each `next` pointer is not evident from the static type. There is also insufficient static type information to disambiguate the current value of an untagged `union` or the size of a dynamically-sized array.

We can overcome these limitations with user-provided annotations. For the purposes of our experiments, we have annotated the embedded list cases (described below), but left the arrays, unions, and other cases to future work; these would probably have annotations similar to those provided by Deputy [21, 109], with the advantage that they would only be required on type definitions and not function

```
// @head, @type super_block(s_list)
struct list_head g_super_blocks;

struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
struct fs_struct {
    struct dentry   *root;
    struct vfsmount *rootmnt;
};
struct dentry {
    struct dentry            *d_parent;
    struct inode             *d_inode;
    struct dentry_operations *d_op;
}
struct inode {
    struct inode_operations *i_op;
    struct file_operations  *i_fop;
    struct super_block      *i_sb;
};
struct super_block {
    // @headed, @type super_block(s_list)
    struct list_head          s_list;
    struct dentry            *s_root;
    struct super_operations *sop;
};
struct vfsmount  {
    struct vfsmount    *mnt_parent;
    struct dentry      *mnt_root;
    struct super_block *mnt_sb;
};
```

Figure 5.4: Simplified Linux type definitions.

declarations. Because we do not annotate arrays, unions, and manufactured or generic (`void*`) pointers, we may not find all reachable function pointers and thus may potentially miss some violations. Nevertheless, even with this limitation we are able to detect all control-modifying rootkits that we could install on our test platform.

We describe our embedded list annotations by example. Consider the simplified `super_block` structure shown in Figure 5.4. Its member `s_list` is of type `struct list_head`, which contains fields that, according to the definition, link to other `list_head` objects. By convention, these other `list_head` objects are actually the `s_list` fields of other `super_block`s, allowing `super_block`s to be chained together into a linked list. Here, the linked list is headed by the global variable `g_super_blocks`, and the last element of the list will point to `g_super_blocks` itself, terminating the list. Traversal code may cast each `next` or `prev` pointer to a `super_block` to access its fields. Of course, such code must check, before performing the cast, that a pointer is to another `super_block` object by ensuring it does not point to `g_super_blocks`, the head/terminator of the list.

So that the monitor can properly traverse embedded lists of objects, we specify this convention using some simple annotations. We annotate each occurrence of a global variable or structure field of type `list_head` with the type of the objects into which the `list_head` actually points. If the pointer is into the middle of a structure, we also include the field name of the precise position; the start of the object can thus be recovered by subtracting `offsetof(`*fname*`)` from the `next` or `prev` pointer. In Figure 5.4, we have added comments including annotation *@type super_block(s_list)* above both the `s_list` member and the `g_super_blocks` global.

Our example illustrates a "headed" list, in which each embedded `list_head` could point to another `list_head` embedded within the given `@type`, or to the head/terminator of the list. We indicate this by annotating a list's head with `@head` (in the example, the `g_super_blocks` global variable is so annotated), while the `list_head`s within a headed list are annotated with `@headed` (in the example, the `s_list` fields are so annotated). To cast such a field to its given `@type` thus requires a check that the field does not point to the head. Alternately, to represent a non-headed list, we can annotate `list_head` occurrences as `@nohead`. This means that they will *always* point to objects of the given `@type`.

For the Linux 2.4.18-3 kernel (part of the default Red Hat 7.3 installation), we annotated 123 type definitions and 39 variable definitions. The process was fairly straightforward and took us just under two days, working sporadically, to complete. We believe that most of these annotations could be inferred — and many of the generic, array, and union annotations as well — using a constraint-based analysis along the lines of CCured [75].

5.3.2.3 Implementing the Traversal

The final step is to use the type graph to generate the traversal code for the monitor. The generated code performs a modified breadth first search (BFS), starting at each global variable whose type appears in the type graph. When an object is visited, all function pointers (if any) that are part of the object are checked, and all neighbors (as determined by the type graph) are added to the queue of nodes remaining to be visited (nodes are marked so they are not revisited). The only exception to BFS ordering occurs when a node is reached that contains one or more list heads annotated with `@head`. In these cases, each list is traversed to completion, following the appropriate `@headed` link field in its members; `@headed` links are ignored except during such traversals. This approach ensures that all members are reached and treated in a type-correct manner. Non-headed list pointers (annotated `@nohead`) are treated like any other neighbor pointer in the graph, processed according to BFS.

Because the traversal will be run within a process outside of the target kernel, it requires a mechanism to map source-level type and variable definitions to their low-level representation in the running kernel. Specifically, two source→binary mappings are required. First, the monitor must know the virtual addresses of the running kernel's global variables. The *Symbol Mapper* (see Figure 5.1) extracts these from the kernel's binary files. Second, the monitor must be able to map source-level types to their binary representations in memory. The *Type Mapper* (implemented in C) uses the kernel compiler to generate this information from the kernel's source-level types.

5.3.2.4 Validating Function Pointers

Once the monitor has located a particular function pointer, it must validate whether the target of that pointer is consistent with the kernel's CFG. We have identified four possible approximations for determining consistency, which we list from least to most precise:

- *Valid code region.* In this approximation, the monitor simply requires that all pointers target some portion of valid code, i.e., the set V calculated during the text validation phase (Section 5.3.1). The performed check is a range comparison within the (small) list of valid ranges.

- *Valid function.* In this approximation, the monitor maintains a list of valid kernel function start addresses for code regions in the set V and requires that all function pointers target one of these addresses. The performed check is set membership in the large set of allowable function pointers.

- *Valid function type.* This approximation narrows the set of functions that a given pointer can target by maintaining a set of valid function addresses for each function pointer type. The performed check is, first, a lookup for the correct set and, second, a set membership check.

- *Valid points-to set.* This approximation utilizes a static or dynamic points-to analysis for each function pointer in the kernel. At runtime, the monitor requires that any encountered function pointer must target one of the functions in its corresponding points-to set. The performed check is the same as in the valid function type case, but the number of sets is likely to be much larger (one per data-structure member, rather than one per-type).

In theory, these approximations could fail to detect an attack that is able to persistently reuse some or all of the kernel's existing functionality (reminiscent of "jump-to-libc"-style attacks [98]). However, we believe that the above approximations will defend against many such attacks because of the difficulty of reusing complete functions for the wrong purpose. Many modern jump-to-libc attacks work by jumping into the middle of code or data that would not be considered valid by our approximations. We have implemented the first two approximations and found that both were sufficient to detect the control-modifying attacks in our test corpus.

5.3.2.5 Monitor Timing

A natural approach to monitoring is to periodically pause the target VM long enough for the monitor process to traverse and validate the target kernel's state. This pause can be disruptive, however; in our benchmarks we have seen the traversal take as long as four seconds. Instead, we could reduce the pause to be just long enough to copy the kernel's memory to the monitor process, where it can be traversed asynchronously. Unfortunately, to do this requires allocating a substantial amount of memory to the monitor process that we would prefer to allocate to the target; the Linux kernel, for example, could occupy up to 1 GB of memory.

To avoid these problems, we allow the monitor process to traverse the target kernel's heap in parallel with the target VM's execution. While better performing, this approach could result in false positives because the monitor may view the kernel's memory inconsistently. For example, the monitor could queue a pointer whose memory is freed by the kernel before the monitor processes it, and thus the monitor will examine stale data. As a result, it may incorrectly conclude that a bogus bit pattern is a valid pointer and follow it, and/or that a bogus bit pattern is an invalid function pointer and complain about it. In the worst case, the monitor could end up traversing stale data indefinitely. Though perhaps less likely, the same problems could arise even from a snapshot taken at a single moment in time, since the paused kernel may be in the middle of a code sequence it assumes will be atomic. For example, it could be in the middle of adding, removing, or initializing an element in a list, and thus the monitor could end up traversing uninitialized or otherwise invalid pointers.

Our monitor implements three safeguards to mitigate problems due to traversing inconsistent states. First, before following a pointer (or validating a function pointer), the monitor confirms that the pointer targets a valid kernel address by consulting the target's page tables. This prevents reading nonsensical or non-kernel data. Second, the monitor places an upper limit on the number of objects traversed to ensure termination. For our experiments (Section 5.4), we utilized an upper limit of 2^{20} objects; the limit was never reached during testing, but has been left in place for safety. For our tests, at most 341,059 objects were encountered on any single pass. Finally, when validating function pointers, our monitor requires the same potential violation (determined by the violating function pointer's address and the address it points to) to be detected in two consecutive monitor runs before raising an alarm. When running at large intervals (currently three seconds or greater), the second "validation" run is commenced within three seconds, rather than waiting for the entire monitor period to expire. This narrows the window between detection and notification, while still allowing the performance tuning to remain in place. We experienced no false positives during any of our experiments using these simple techniques.

5.4 Experiments

In this section, we present the results of a series of experiments performed using our VMM-based SBCFI monitor. We used the Xen virtual machine monitor, described in Chapter 4, for our test platform because it could run unmodified kernels on which we could install a large percentage of the collected attacks.

5.4.1 Detection

To demonstrate the effectiveness of SBCFI at detecting kernel attacks, we collected as many publicly-available rootkits as we could find and tested them on our target platform (see Chapter 2). Of the 25 that we acquired, we were able to install 18 in our virtual test infrastructure. The remainder either did not support our test kernel version or would not install in a virtualized environment. We installed the rootkits using one of two mechanisms – malicious LKM loading or injection via `/dev/kmem`, a virtual device object that gives direct access to kernel virtual memory.

Rather than use the most recent version of the Linux kernel (2.6), we instead performed our tests using an older (2.4) version, which is vulnerable to the majority of our analyzed attacks. The alternative strategy would have required porting many of the attacks to a newer kernel. Therefore, the protected kernel for all tests was Linux 2.4.18-3, the default kernel for RedHat Linux 7.3 installations. No modifications were made to this kernel for testing. While generating our monitor for this kernel, our source-code analysis tools extracted 1049 types and 22182 global names, 5105 of which were functions. Based on the calculated type graph, we identified 660 roots: 400 globals contain function pointers, while an additional 260 are viable starting points for reaching function pointers somewhere in memory.

In our tests, we successfully detected all of the attacks that make persistent modifications to the kernel's control-flow. For loaded LKMs, our monitor detected both the existence of an untrusted module and any function pointer references to that module. Direct-injection attacks and those module attacks that remove themselves from the module list for stealth were detected when a persistent pointer targeted the injected code. On several occasions, our monitor also detected the transient changes introduced as part of the direct injection attacks (in addition to their persistent changes). The only attack not detected by SBCFI, `hp`, makes no persistent control-flow modifications, as described in Section 2.2.3. We encountered no false positives during any of our benchmarks or detection tests.

At runtime, our monitor visited, on average, 82,097 objects and validated 39,278 function pointers per iteration. The average runtime per iteration was 624 milliseconds for our two-CPU configuration and 1.78 seconds for one CPU. We found that these statistics varied greatly depending on system load, but they never exceeded 341,059 nodes visited and 233,197 function pointers. Maximum traversal time was as long as 4.85 seconds for the two-CPU configuration and as long as 8.2 seconds for one CPU. In general, the number of objects visited increased with system uptime.

5.4.2 Performance

To evaluate the overhead imposed by SBCFI monitoring, we measured the performance of the target VM using three benchmarks: the integer workloads from the SPECCPU2006 benchmark suite [99], HP's Netperf networking microbench-

Target Hardware Configuration	
Machine Type:	Dell Precision 490 Workstation
Processor:	Intel Xeon Quad-core X5355, 2.66GHz
RAM:	4GB
Storage:	160GB SATA
Networking:	Broadcom NetXtreme BCM5752 Gigabit
Target Software Configuration	
Version:	Xen-3.1.0
Host OS:	Debian Etch, full installation
	Linux 2.6.18-5-686 kernel
Guest OS:	RedHat Linux 7.3 full installation
Guest memory:	1200M

Table 5.1: SBCFI test platform summary.

mark [44], and Linux kernel build time. Additionally, we ran each test in both one-CPU and two-CPU configurations. In the one-CPU configuration, all virtual machines and the VMM shared a single processor. In the two-CPU configuration, the monitor's VM and the target VM could utilize separate processors simultaneously.

For the Netperf networking experiments, we ran Netperf's default TCP stream test, which measures TCP throughput for a single connection over a fixed period of time. Described in Table 5.2, we used an additional system to act as the data-generating sender in the Netperf tests. We tested three different socket buffer sizes (both client and server were set to the same send/receive size), as shown in Table 5.2. Each test was performed 30 times per buffer size, and we ran each test for a number of monitoring periods, ranging from no SBCFI monitor (i.e., just a VM) to near constant monitoring.

In addition to the networking benchmarks, we tested the integer workloads

Client Hardware Configuration	
Machine Type:	Lenovo Thinkpad T60
Processor:	Intel CoreDuo T2500, 2GHz
RAM:	1.5GB
Storage:	80GB IDE
Networking:	Built-in Intel 82573L Gigabit
Client Software Configuration	
OS:	Gentoo Linux Base 1.12.9 Linux 2.6.20.1 kernel
Test Parameters	
Netperf Version:	2.4.3
Network:	Gigabit Ethernet switch, Cat5e cable
Time Per Run:	60secs
Number of Runs:	30 per socket size
Socket Sizes:	128KB, 56KB, 8KB

Table 5.2: Netperf TCP stream test platform summary.

(referred to collectively as SPECINT2006) of the SPECCPU2006 benchmark suite on the target system. Like the Netperf benchmarks, we ran these tests both with and without the monitor for one-CPU and two-CPU configurations. The same target machine, summarized in Table 5.1, was used for these tests.

Because Netperf and SPECINT2006 are microbenchmarks aimed at testing specific aspects of system performance, we utilized Linux kernel builds as an application benchmark to demonstrate how SBCFI impacts performance during a typical system workload. A combination of both disk I/O and CPU-intensive compilations, kernel builds are common tasks that are routinely performed by system administrators. In our kernel build tests, the Linux 2.4.32 kernel was unpacked, configured, built (with command line `make bzImage modules`), and removed in a loop. We performed this series of operations 30 times for each tested configuration, recording the time of the build stage (not including unpacking, configuration, or removal). Because we ran our kernel build tests using a looping script, we found that the target system's buffer cache typically sped up kernel builds after the first build, resulting in differences of several seconds between the first build and the remaining builds. Therefore, to simulate the first build's conditions for each iteration, we added an additional unmount/mount stage before our timed build stage. In this new step, we unmounted and then re-mounted the partition where the kernel source was configured, causing the buffer cache to clear itself before the build stage began.

5.4.2.1 Results

Table 5.3 shows the median throughput for 30 Netperf test runs, which we performed for each of the listed configurations. There are two types of configurations shown in every sub-table, which each present data for a particular socket buffer size (128KB, 56KB, and 8KB for tables (a), (b), and (c) respectively). The first row of the sub-table shows the test results for our target VM without any SBCFI monitoring. The remaining four rows describe tests in which the measured Xen guest was being monitored by our SBCFI implementation, which was configured to run asynchronously at various periods (every second, every five seconds, etc.). The second column provides the median throughput for 30 runs in each configuration, along with the semi-interquartile range (SIQR, shown in parentheses).[3] The third column shows the incurred overhead in comparison with the non-monitor (row one)

[3]The *semi-interquartile range* (SIQR) is the difference between the high and low quartiles divided by two.

Configuration	Throughput in 10^6 bits/sec (SIQR)	Penalty
Xen 3.1.0	177.8 (0.3)	0%
SBCFI (1s)	150.4 (0.5)	15.4%
SBCFI (5s)	173.3 (0.5)	2.5%
SBCFI (10s)	176.3 (0.3)	0.8%
SBCFI (15s)	177.4 (0.2)	0.2%

(a) 128K socket size

Xen 3.1.0	177.3 (0.2)	0%
SBCFI (1s)	150.9 (0.3)	14.9%
SBCFI (5s)	173.9 (0.6)	1.9%
SBCFI (10s)	176.5 (0.3)	0.5%
SBCFI (15s)	177.5 (0.2)	-0.1%

(b) 56KB socket size

Xen 3.1.0	118.1 (0.1)	0%
SBCFI (1s)	99.2 (0.1)	16.0%
SBCFI (5s)	114.3 (0.2)	3.3%
SBCFI (10s)	116.4 (0.1)	1.5%
SBCFI (15s)	117.2 (0.0)	0.8%

(c) 8KB socket size

Table 5.3: Netperf 2CPU TCP throughput results.

configuration. Table 5.4 is formatted the same as Table 5.3, except that the data reflects a one-CPU system configuration.

Our Netperf results show that SBCFI imposes minimal impact when monitoring at periods of 5 seconds or greater — at most 3.3% in any configuration at that period. Additionally, SBCFI's impact can be scaled to under 1% by increasing the monitoring period; monitor periods under 15 seconds were sufficient for all cases. While our results show small performance improvements for some cases (specifically, some 10- and 15-second tests), these results are explainable within the variation suggested by the SIQR.

While we have shown that SBCFI's impact is minimal and scalable, we would

Configuration	Throughput in 10^6 bits/sec (SIQR)	Penalty
Xen 3.1.0	139.3 (0.7)	0%
SBCFI (1s)	123.2 (0.7)	11.6%
SBCFI (5s)	135.0 (0.6)	3.1%
SBCFI (10s)	137.2 (0.5)	1.6%
SBCFI (15s)	140.7 (0.8)	-0.9%

(a) 128K socket size

Xen 3.1.0	139.0 (0.5)	0%
SBCFI (1s)	123.3 (0.4)	11.3%
SBCFI (5s)	134.9 (0.6)	3.0%
SBCFI (10s)	137.2 (0.7)	1.3%
SBCFI (15s)	140.9 (0.7)	-1.3%

(b) 56KB socket size

Xen 3.1.0	95.5 (0.2)	0%
SBCFI (1s)	95.5 (0.5)	0%
SBCFI (5s)	95.4 (0.3)	0.2%
SBCFI (10s)	95.6 (0.3)	-0.1%
SBCFI (15s)	96.4 (0.2)	-0.9%

(c) 8KB socket size

Table 5.4: Netperf 1CPU TCP throughput results.

also like to determine the amount of overhead imposed by the Xen VMM itself, since it contributes costs that may not otherwise exist, e.g., if the hypervisor were deployed only to enable kernel monitoring. Because our target VM is a fully virtualized Xen guest, I/O operations incur additional overhead due to more hypervisor traps that lead to I/O operations performed in domain 0 on behalf of the guest. In many cases, this overhead can be quite significant. To address these issues, the Xen developers have implemented a set of special-purpose drivers that directly implement paravirtual operations for fully virtualized guests. These drivers are referred to as paravirtualized (PV) drivers and significantly improve I/O performance. Unfortunately, PV drivers are not available for the version of the kernel that we used

Configuration	Throughput in 10^6 bits/sec (SIQR)	Penalty
Native (no VMM)	800.8 (0.8)	–
Xen 3.1.0, no PV drivers	132.9 (0.4)	83.4%
Xen 3.1.0, PV drivers	798.3 (2.6)	0.3%

(a) 128K socket size

Native (no VMM)	794.6 (0.8)	–
Xen 3.1.0, no PV drivers	139.3 (0.5)	82.5%
Xen 3.1.0, PV drivers	797.7 (2.4)	-0.4%

(b) 56KB socket size

Native (no VMM)	466.2 (0.1)	–
Xen 3.1.0, no PV drivers	109.1 (0.6)	76.6%
Xen 3.1.0, PV drivers	341.7 (0.5)	26.7%

(c) 8KB socket size

Table 5.5: Netperf TCP throughput VMM overhead.

for our SBCFI tests. However, we performed an additional set of tests using a more recent kernel, both with and without PV drivers, to gain a sense of this overhead.

Table 5.5 shows the results of three additional runs of our Netperf experiments. In the first row, we have configured our test system with a native GNU/Linux Debian 4.0 distribution (i.e., without Xen or a guest VM). Row two shows the tests for the same Debian distribution, as installed on a fully-virtualized Xen guest, similar to row 1 of Tables 5.3 and 5.4. Finally, row three shows results of the same Debian VM with Xen's custom PV drivers installed.

From the results of our additional Xen Netperf experiments, we make two observations. First, Xen's overhead is significant without PV drivers – as much as six times worse than native performance. Second, the use of PV drivers dramatically improves the performance of the guest to near native values in most cases. Throughput for the 8KB socket size improved less than for the bigger socket sizes (although

Configuration	**Time (s)**	**% Native**	**% VMM**
Native	9956	0%	—
Xen 3.1.0	10610	6.6%	0%
SBCFI (Cont.)	10796	8.4%	1.8%
SBCFI (1s)	10687	7.3%	0.7%
SBCFI (3s)	10634	6.8%	0.2%

Table 5.6: SPECINT2006 2CPU median run times.

Configuration	**Time (s)**	**% Native**	**% VMM**
Native	9954	0%	—
Xen 3.1.0	10883	9.3%	0%
SBCFI (5s)	12297	23.6%	13.0%
SBCFI (10s)	11649	17.0%	7.0%
SBCFI (30s)	11161	12.1%	2.6%
SBCFI (60s)	11024	10.8%	1.3%
SBCFI (90s)	10973	10.2%	0.8%

Table 5.7: SPECINT2006 1CPU median run times.

still dramatically). While we cannot directly account for this disparity, likely causes include implementation details of the PV drivers or the added overhead caused by more receive packet interrupts at the smaller size. The apparent small improvement over native performance for the 56KB socket PV case can be explained by sampling variation, as indicated by the SIQR values.

Table 5.6 shows the results of the SPECINT2006 tests for the two-CPU configuration. Unlike the I/O experiments, there are no special drivers to improve CPU performance. Therefore, we have included the native hardware results in the same table. The first column of Table 5.6 indicates which configuration was under test. The second column shows the time in seconds for the median of three runs of the twelve SPECINT2006 workloads. The third and fourth columns show overhead relative to the raw hardware (row 1) and VMM-only (row 2) configurations respectively.

Configuration	**Time in seconds (SIQR)**	**% Native**	**% VMM**
Native	101.5 (0.2)	0%	—
Xen 3.1.0	113.4 (0.8)	11.8%	0%
SBCFI (1s)	115.6 (1.4)	13.9%	1.9%
SBCFI (5s)	115.3 (1.1)	13.6%	1.6%
SBCFI (10s)	114.2 (0.8)	12.6%	0.7%
SBCFI (15s)	113.9 (0.7)	12.2%	0.4%

Table 5.8: Linux kernel build 2CPU median run times.

Configuration	**Time in seconds (SIQR)**	**% Native**	**% VMM**
Native	101.5 (0.2)	0%	—
Xen 3.1.0	118.5 (1.1)	16.8%	0%
SBCFI (1s)	169.4 (1.1)	66.9%	43.0%
SBCFI (5s)	132.8 (0.7)	30.1%	12.1%
SBCFI (10s)	125.3 (0.6)	23.4%	5.8%
SBCFI (20s)	122.1 (0.9)	20.3%	3.1%
SBCFI (30s)	119.9 (0.7)	18.1%	1.2%
SBCFI (45s)	119.4 (1.0)	17.6%	0.8%

Table 5.9: Linux kernel build 1CPU median run times.

Table 5.7 is similar to Table 5.6, but shows the results for our one-CPU setup.

The SPECINT2006 experiments show that the majority of the overhead is the result of the VMM and not SBCFI, except for the one-CPU configuration, where SBCFI imposes up to 13% penalty when operated at 5-second intervals. In all cases, SBCFI itself imposes a tunable penalty, trading off precision for performance, on top of the VMM, with unnoticeable overhead when monitoring at 1-second intervals for the two-CPU configuration and 90-second intervals for the one-CPU configuration.

Tables 5.8 and 5.9 show the results of our Linux kernel build application benchmarks and are formatted identically to our SPECINT2006 results. As with the CPU benchmarks, we have included comparisons with native performance in the same table and did not use PV drivers for any of these tests. The median time (and SIQR) for 30 builds is reported for each configuration.

The results of our Linux kernel build experiments again show the scalable performance impact of SBCFI. For two-CPU configurations, SBCFI imposed less than 2% impact in all tests, with less than 1% overhead when running at intervals of 10 seconds or greater. For one-CPU configurations, 45-second intervals achieved the same. Additionally, we again found that Xen itself was responsible for the majority of the overhead in the system. Note that the VMM's impact was less than the impact measured for Netperf (without PV drivers), an I/O-intensive test, and greater than the impact measured for SPECINT2006, a CPU-intensive workload. This makes sense, given that kernel builds require a mix of both types of operations.

5.4.2.2 Discussion

Our experiments show that the overhead of SBCFI on its own is quite small, and that the primary cause of overhead is due to the VMM itself, particularly when multiple processors are available. We do not believe this reflects poorly on SBCFI itself for two reasons.

First, the VMM overhead is a function of the VMM we used, not of VMM technology in general. As described previously, we chose the VMM configuration of our test platform to maximize threat testing rather than performance. We ran Xen in its fully virtualized (as opposed to paravirtualized) mode to support an unmodified kernel on which we could use unmodified attacks. As we have shown, Xen can achieve better performance for fully virtualized hosts with the use of special drivers (around 2% of native on average [105]), and there is nothing that precludes SBCFI monitoring of either paravirtualized or fully virtualized hosts using these drivers. The increasing deployment of high-performance virtualization solutions [9] provides further evidence that VMMs can have reasonable performance when compared to raw hardware.

Second, as described in Chapters 4 and 7, other low-level monitor mechanisms are available, some of which may result in lower monitor overhead. However, we have not yet tested SBCFI using another monitor platform.

In short, our experiments show that SBCFI is effective and practical, detecting all of the kernel attacks we could install on our platform while imposing minimal impact on top of the VMM.

Chapter 6

Specification-Based Checking of Non-Control Data

While kernel control-flow integrity is a useful property to enforce, it is not sufficient to prevent or detect all forms of kernel attacks. As described in Chapter 2, some attacks work by modifying only the kernel's non-control data and allowing its unmodified code to operate on low-integrity data. We refer to such attacks as *non-control data attacks* [18].

In this chapter, we describe the second of two property enforcement modules that we have developed. This module takes as input a manually-produced *specification* of one or more properties of the kernel's state and ensures that those properties hold during execution. Our specification language allows an expert to describe, in a simple but precise way, how kernel objects relate to one another in memory, as well as a set of properties that must hold on those data objects for the integrity of the kernel to remain intact.[1] The input specification is automatically compiled into a series of low-level checks that enforce the desired properties as part of a state-based kernel monitor. The result is a system that allows experts to concentrate on high-level concepts, such as identifying security-relevant non-control data properties, rather than writing low-level code to analyze kernel data structures. While not a complete solution to the problem of kernel data-only attacks, our approach pro-

[1]The specification system described in this chapter was developed jointly with Tim Fraser, AAron Walters, and Bill Arbaugh [84].

vides a flexible short-term complement to the SBCFI property enforcement module described in the previous chapter.

To evaluate the feasibility of a specification-based approach, we have designed a custom property language that describes a high-level model made up of sets of kernel objects and relations among those sets. In our system, a specification is comprised of a list of rules that map low-level data to a high-level model, as well as a list of property rules that must hold for the model. Additionally, we have implemented a compiler for our language as a module in our property enforcement infrastructure. The compiler transforms expert-written specifications into low-level checks for the Linux kernel that are enforced by our state-based monitor.

To demonstrate the effectiveness of our module, we have written specifications for detecting two data-only attacks. Our first specification detects process hiding attacks, such as those implemented by the `hp` rootkit, described in Section 2.2.3. Our second example specification detects a simulated data-only attack against the SELinux access vector cache (AVC). Our experiments show that code produced by our compiler for both specifications successfully detects the target attacks, with no false positives experienced during our tests.

6.1 Writing Specifications: a Linux Hidden Process Example

In this section, we utilize the example of the `hp` Linux kernel rootkit to introduce our example specification language. As described in Section 2.2.3, `hp` performs process hiding through non-control data modifications and is therefore not detected by SBCFI. As a convenience, rather than inventing our own language entirely, we have chosen to adapt a data structure specification language created by Demsky and Rinard [25] for performing specification-based data structure repair. For the sake of clarity, we delay direct comparison between the two languages until Section 6.4.

Our specification language is divided into four parts, each with its own syntax, corresponding to the four tasks that must be performed by the specification writer: (1) describe (or extract) the format of low-level data in kernel memory; (2) declare a high-level model to represent abstractions on the low-level data; (3) define a list of rules that map the low-level data to the high-level model; and (4) define the properties that must hold in the high-level model if the kernel's integrity is maintained.

The example specification presented in this section enforces a simple property for detecting data-only process hiding in Linux (Section 2.2.3). The basic strategy is to require all processes that exist in Linux's process scheduling list to also appear in the kernel's all-tasks list. The four components of the specification are summarized as follows: (1) describe the low-level types for Linux's process accounting data structures; (2) declare sets in a high-level model representing the set of currently scheduled tasks and the set of all tasks; (3) define rules for traversing the kernel's lists and defining corresponding members of the declared sets; and (4) specify a property of the generated sets that requires all members of the scheduled set to also be members of the all-tasks set. We describe each part of the specification in turn, moving from the most concrete to the most abstract.

Low-Level Structure Definition: The first part of our specification language provides C-like constructs for describing the layout of low-level objects in memory. Our structure definitions provide a few additions to the normal C language syntax. First, fields may be marked "reserved," indicating that they exist but are not used in the rest of the specification. Second, array lengths may be variable and determined at runtime through expression evaluation. Third, a form of structure "inheritance" is provided for notational simplicity, whereby structures can be defined based on other structures and then expanded with additional fields. Figure 6.1(a) contains our specification of the Linux kernel's process accounting data structures, written in the structure definition syntax.

The compiler translates these structure definitions into to C structure definitions, along with a set of access functions for reading the structures from the target kernel's memory. Figure 6.1(b) shows the result of translating the example specification's structure definitions into the corresponding C declarations for use in the monitor's code. Note the use of the `host_addr_t` type to represent host addresses for pointers. This abstraction is necessary for monitors, such as coprocessors, which may have different processor byte orders or address sizes. Similarly, many virtual machine monitors support both 32-bit and 64-bit guests. As a result, code running in the monitor VM may not have the same pointer size as the target kernel.

As a final note about structure definitions, they may be eliminated from the specification entirely when kernel source code is available, as is the case for our Linux kernel experiments. When source is available, we can make use of the automated type and global extraction infrastructure described in Section 5.3.2. However, the

```
Task init_task;

structure Task {
        reserved byte[32];
        ListHead run_list;
        reserved byte[52];
        ListHead tasks;
        reserved byte[52];
        int pid;
        reserved byte[200];
        int uid;
        reserved byte[60];
        byte comm[16];
}

structure ListHead {
        ListHead *next;
        ListHead *prev;
}

sructure Runqueue {
        reserved byte[52];
        Task *curr;
}
```

(a) Low−Level Structure Definiton

```
host_addr_t init_task =
    LINUX_SYMBOL_init_task;
struct Task {
        unsigned char reserved_1[32];
        ListHead run_list;
        unsigned char reserved_2[52];
        ListHead tasks;
        unsigned char reserved_3[52];
        int pid;
        unsigned char reserved_4[200];
        int uid;
        unsigned char reserved_5[60];
        unsigned char comm[16];
};

struct ListHead {
        host_addr_t next;
        host_addr_t prev;
};

struct Runqueue {
        unsigned char reserved_1[52];
        host_addr_t curr;
};
```

(b) Translated Structure Definiton

```
set AllTasks(Task);

set RunningTasks(Task);
```

(c) Model Space Declarations

```
[ for_circular_list i as ListHead.next starting init_task.tasks.next ], true −> container(i, Task,tasks.next) in AllTasks;
[ ], true −> runqueue.curr in RunningTasks;
```

(d) Model Building Rules

```
  [ for t in RunningTasks ],  t  in  AllTasks
        : notify_admin("Hidden task " + t.comm + " with PID " + t.pid + " detected at kernel virtual address " + t);
```

(e) Property Checking Rules

Figure 6.1: Process accounting subsystem specification.

structure definition language is useful for proprietary kernels, where source code is not available, and is helpful for understanding the rest of our example.

Model Space Declaration: The second part of a specification, shown in Figure 6.1(c) for our process accounting example, declares a group of sets or relations (there are no relations in our first example) that exist in the high-level model. There are two sets in our specification: one corresponding to all processes in the all-tasks list (the `AllTasks` set) and one corresponding to all processes in the run queue (the `RunningTasks` set). Both are of type `Task` in the model. Sets are composed of objects defined in the structure definition language, while relations describe mappings between sets.

Model Building Rules: The model building rules bridge the gap between the low-level types and the model space declarations by identifying which low-level objects should be used within the abstract model. That is, model building rules provide definitions for the sets and relations declared in the model space declarations. These rules take the form

```
[<quantifiers>], <guard> -> <inclusion rule>;
```

For each rule, there is a set of quantifiers that enumerates the objects to be processed by the rule, a guard that is evaluated for each quantified object to determine if it should be subject to the rule, and an inclusion that determines how that object should be classified in the abstract model. Examples of common quantifiers include:

- `for_list`: Linked lists are a common programming paradigm. This expression gives specification writers a straightforward way to indicate that they intend to traverse a list up to the provided stop address (or NULL if not indicated).

- `for_circular_list`: This is syntactic sugar for the `for_list` construct where the end address is set equal to the first object's address. The Linux kernel makes heavy use of circular lists.

- `for n in N`: This expression gives a mechanism for referring to all objects `n` in the set `N`.

- `for i = a to b`: This expression provides numerical loop iteration for an index `i` over the integers [`a`, `b`).

Guards specify predicates over the quantified objects. Examples include numerical comparison operators (`=`, $<$, and $>$), the logical operators `AND` and `OR`, and the literal `true`.

Inclusions are very straightforward in that they simply specify which set or relation the quantified elements belong in. As described in Section 5.3.2.2, a common programming paradigm (especially in the Linux kernel) is to embed generic list pointer structures as members within another data structure. Our `container()` expression gives specification writers an easy way to identify the object of which a particular field is a member. This extension is not necessary in cases where source code annotations for embedded lists have already been applied to data types.

Figure 6.1(d) shows the model rules for our process accounting example. The first rule indicates that a circular list starting (and ending) at `init_task.tasks.next` will be processed. The keyword `true` in the guard indicates that all members of this list should be subject to the inclusion. The inclusion itself uses our `container()` expression to locate the `Task` that contains the list pointer and to include that `Task` in `AllTasks`. The second rule is very simple; it creates a singleton set `RunningTasks` with the current task running on the run queue.

Property Checking Rules: The final part of the specification defines a set of properties that are expected to hold for the high-level model. These rules take the following form:

```
[ <quantifiers> ], <predicate> :  <[consistency,] response>;
```

In property checking rules, the set of quantifiers may include only sets defined in the model. The predicate is evaluated on each quantified member and may include set operations and evaluations of any relations defined in the model. If the predicate fails for any quantified member, the action specified by the `response` portion of the rule is carried out. This term allows the specification writer to dictate how failures are to be handled for a particular rule.

One challenge facing low-level property checking in our system stems from the possibility that our monitor is operating asynchronously with the kernel. As described in Section 5.3.2.5, the monitor may read inconsistent data, e.g., when the kernel is in the middle of updating some of its data structures but has not yet finished. Invalid reads have the potential to produce inconsistent high-level models, which may cause property checks to incorrectly fail (i.e., cause false positives).

As a mitigation strategy for inconsistent data reads, the property checking rule's response term allows for an optional "consistency parameter." This parameter allows the specification writer to identify a "safe" number of failures for a given rule before the response action is performed, similar to the re-validation performed by our SBCFI module (see Section 5.3.2.5). If no such parameter is provided, the default value of two consecutive failures is used. Of course, a secondary result is that actual rule violations will be given an opportunity to occur once without detection. The specification writer will need to balance the advantages and disadvantages for each rule and can always disable this feature by setting the value to zero. For the threat considered in our Linux process accounting example, the default value is acceptable because of the nature of the targeted threat. A process that is short-lived has no reason to hide, since an administrator is unlikely to notice the process.

Figure 6.1(e) shows the single property check for our hidden process example. The rule states that if any process is ever found running on the processor that is not in the all-tasks list, we have a security problem and need to alert the administrator. This example describes a relatively simple method of detecting hidden processes. To detect a hidden process, the monitor must catch the process while it has the host CPU — a probabilistic strategy that is likely to require many samples of the host's state over time before the hidden process's luck runs out. A more deterministic approach would be to compare the members of the kernel's numerous wait and run queues with the members of the all-tasks list. To be eligible for scheduling, a process must be on one of these wait or run queues; a process on a wait or run queue but not in the all-tasks list is hiding. This strategy would require a more complex model specification.

6.2 A Second Example: the SELinux AVC

To further demonstrate the effectiveness of our specification-based approach, we provide an additional example that utilizes a more complex specification. Our second specification is useful for protecting access control decisions performed by the SELinux security subsystem of the Linux kernel. We begin with a brief introduction to SELinux and its access vector cache before describing a simulated attack against the AVC that allows attackers to surreptitiously elevate privileges of their userspace processes. Finally, we present an example specification for detecting this attack.

When most actions occur in the kernel, some form of a capability is used to identify whether or not a principal should be given (or already has been given) access to a resource. These capabilities therefore represent a prime target for attackers wishing to elevate privilege. Changing process user identifiers (UIDs) has long been a favorite technique of attackers. Other examples include file descriptors and sockets (both implemented in the same abstraction in the kernel). The SELinux access vector cache provides a good example of this kind of capability and represents a potential target for an adversary seeking privilege escalation. We now describe the structure and purpose of the AVC and how an adversary might tamper with its state.

SELinux [62] is a security module for Linux kernels that implements a combination of Type Enforcement [11] and Role-based [32] mandatory access control, now included in some popular GNU/Linux distributions. During runtime, SELinux is responsible for enforcing numerous rules governing the behavior of processes. For example, one rule might state that the DHCP [29] client daemon can only write to those system configuration files needed to configure the network and the Domain Name Service [68], but no others. By enforcing this rule, SELinux can limit the damage that a misbehaving DHCP client daemon might cause to the system's configuration files should it be compromised by an adversary (perhaps due to a buffer overflow or other flaw).

To enforce its rules, SELinux must make numerous decisions during runtime such as "Does the SELinux configuration permit this process to write this file?" or "Does it permit process A to execute program B?" Answering these questions involves some overhead, so SELinux includes a component called the access vector cache to save these answers. Whenever possible, SELinux rapidly retrieves answers from the AVC, resorting to the slower method of consulting the policy configuration only on AVC misses.

On our experimental system, the AVC is configured to begin evicting least frequently-used entries after reaching a threshold of 512 entries. Our single-user system never loaded the AVC much beyond half of this threshold — although it was occasionally busy performing builds, these builds tended to pose the same small number of access control questions again and again. However, one could imagine a more complex multi-user system that might cause particular AVC entries to appear and disappear over time. Installations that permit SELinux configuration changes during runtime might also see AVC entries evicted due to revocation of privileges.

SELinux divides all resources on a system (such as processes and files) into distinct classes and gives each class a numeric Security Identifier or "SID." It expresses its mandatory access rules in terms of what processes with a particular SID may and may not do to resources with another SID. Consequently, at a somewhat simplified abstract level, AVC entries take the form of tuples:

```
<ssid, tsid, class, allowed, decided, audit-allow, audit-deny>
```

The `ssid` field is the SID of the process taking action, the `tsid` field is the SID of the resource upon which the process wishes to act, and the `class` field indicates the kind of resource (file, socket, and so on). The `allowed` field is a bit vector indicating which actions (read, write, and so on) should be allowed and which should be denied. Only some of the `allowed` field bits may be valid — for example, if the questions answered by SELinux so far have involved only the lowest-order bit, then that may be the only bit that contains a meaningful 0 or 1. SELinux may or may not fill in the other `allowed` field bits until a question concerning those bits comes up. To distinguish a 0 bit indicating "deny" from a 0 bit indicating "invalid," the `decided` field contains a bit vector with 1 bits for all valid positions in the `allowed` field. The `audit-allow` and `audit-deny` fields are also bit vectors; they contain 1 bits for operations that should be logged to the system logger when allowed or denied, respectively.

It is conceivable that adversaries who have already gained administrative control over a system might wish to modify the SELinux configuration to give their processes elevated privileges. Certainly, they could accomplish this most directly by modifying the SELinux configuration files, but filesystem integrity monitors like Tripwire [56] would easily detect such modifications. Alternatively, adversaries might modify the in-kernel data structures representing the SELinux configuration — the same data structures SELinux consults to service an AVC miss. However, these data structures change infrequently, when administrators decide to modify their SELinux configuration during runtime. Consequently, any tampering might be discovered by a traditional kernel integrity monitor that performs hashing or makes comparisons with correct, known-good values.

The state of the AVC, on the other hand, is dynamic and difficult to predict at system configuration time. Entries come and go with the changing behavior of processes. An adversary might insert a new AVC entry or modify an old one to effectively add a new rule to the SELinux configuration. Such an entry might add extra `allowed` and `decided` field bits to grant additional privileges, or remove existing `audit-allow` and `audit-deny` field bits to turn off troublesome logging. Such an entry would override the proper in-memory and on-disk SELinux configuration for as long as it remained in the cache. On a single-user installation like our experimental system, it would face little danger of eviction. On a busier system, frequent use might keep it cached for as long as needed.

6.2.1 SELinux AVC Verification

Our approach for protecting the AVC begins with the assumption that a simple "binary" integrity check is protecting the static data structures that represent the full SELinux policy. We then use our specification enforcement module to implement a specification whose goal is to compare all AVC entries with their protected entries in the full policy. Figures 6.2 and 6.3 display the complete specification we used to protect the SELinux AVC. This specification is more complex than the previous example, largely due to the complexities of the SELinux system and its data structures. However, the complexity of the specification is minimal as compared with the number of lines of code that would be required to implement the equivalent checks in low-level code (eight model building rules and one property check rule versus the 709 lines of C code in our example hand-coded implementation).

```
 SidTab sidtab;
 AVCCache avc_cache;
 Policydb policydb;

structure ListHead {
        ListHead *next;
        ListHead *prev;
}
structure AVCCache {
        ListHead slots[512];
}

structure Policydb {
        reserved byte[108];
        AVTab te_avtab;
        rserved byte[8];
        AVTab te_cond_avtab;
}
```

```
structure AVCNode {
        int ssid;
        int tsid;
        short tclass;
        reserved short;
        int allowed;
        int decided;
        int auditallow;
        int auditdeny;
        int seqno;
        int atomic;
        ListHead list;
}
```

```
structure AVTab {
        AVTabNode **htable;
        int nel;
}

structure AVTabNode {
        int source_type;
        int target_type;
        int target_class;
        int specified;
        int allowed;
        int auditdeny;
        int auditallow;
        AVTabNode *next;
}
```

```
structure SidTab {
        SidTabNode **htable;
        int nel;
}

structure SidTabNode {
        int sid;
        int user;
        int role;
        int type;
        reserved byte[24];
        SidTabNode *next;
}
```

(a) Low−Level Structure Definitons

```
Set AllSids(SidTabNode);                 avcssidtype : AllAVCNodes −> AllSids;
Set AllAVCNodes(AVCNode);                avctsidtype : AllAVCNodes −> AllSids;
Set TEAVTabNodes(AVTabNode);             avcteavtabmapping : AllAVCNodes −> TEAVTabNodes;
Set TECondAVTabNodes(AVTabNode);         avctecondavtabmapping : AllAVCNodes −> TECondAVTabNodes;
```

(b) Model Space Declarations

Figure 6.2: SELinux access vector cache structure and model declarations.

There are four primary entities in our SELinux specification: the security identifier table (of type SIDTab), the access vector cache (an AVCCache), the Type Enforcement access vector table (an AVTab), and its counterpart the Type Enforcement conditional access vector table (also an AVTab). The model building rules first create a set of SIDs by walking through the SID table and then, similarly, create a set of all AVC nodes from the AVC. The third and fourth rules are used to create mappings between the AVC nodes and their source and target SIDs. Rules five and six look-up each AVC node in the full Type Enforcement policy for both conditional and non-conditional access vector tables. The final two model building rules create a mapping between AVC nodes and their corresponding entries in the Type Enforce-

```
[ for i = 0 to 128, for_list j as SidTabNode.next starting sidtab.htable[i] ], true −> j in AllSids ;
[ for i = 0 to 512, for_circular_list j as ListHead.next starting avc_cache.slots[i] ], true −>
        container (j, AVCNode, list.next ) in AllAVCNodes ;
[ for a in AllAVCNodes, for s in AllSids ], (a.ssid = s.sid) −> <a,s> in avcssidtype ;
[ for a in AllAVCNodes, for s in AllSids ], (a.tsid  = s.sid) −> <a,s> in avctsidtype ;
[ for a in AllAVCNodes, for_list j as AVTabNode.next starting
      policydb.te_avtab.htable[(a.tclass + a.avctsidtype.type * 4 + a.avcssidtype.type * 512) & 32767]],
      (j.source_type = a.avcssidtype.type AND j.target_type = a.avctsidtype.type)
              −> j in TEAVTabNodes;
[ for a in AllAVCNodes, for_list j as AVTabNode.next starting
      policydb.te_cond_avtab.htable[(a.tclass + a.avctsidtype.type * 4 + a.avcssidtype.type * 512) & 32767]],
      (j.source_type = a.avcssidtype.type AND j.target_type = a.avctsidtype.type)
              −> j in TECondAVTabNodes;
[ for c in AllAVCNodes, for a in TEAVTabNodes ],
       (c.avcssidtype.type = a.source_type AND
        c.avctsidtype.type = a.target_type AND
        c.tclass = a.target_class) −>
           <c,a> in avcteavtabmapping;
[ for c in AllAVCNodes, for a in TECondAVTabNodes ],
       (c.avcssidtype.type = a.source_type AND
        c.avctsidtype.type = a.target_type AND
        c.tclass = a.target_class) −>
           <c,a> in avctecondavtabmapping;
```

(a) Model Building Rules

```
[ for c in AllAVCNodes ], c.allowed = (c.avcteavtabmapping.allowed | c.avctecondavtabmapping.allowed)
       : notify_admin ("AVC Cache entry has improper privileges " + c.allowed + " at virtual address " + c);
```

(b) Property Checking Rules

Figure 6.3: SELinux access vector cache specification rules.

ment access vector tables. The single property checking rule simply walks through all AVC nodes and checks that the `allowed` field matches the combined (bitwise OR) value of the two corresponding Type Enforcement access vector entries for that AVC node. As with the last example, the monitor notifies an administrator if the data structures are found to be inconsistent.

We have tested our code against an attacking loadable kernel module that modifies the permissions for a particular AVC entry. A rootkit might make such a modification to elevate temporarily the privileges of one or more processes in a manner that could not be detected by an integrity monitor that observed only static data structures. Our specification successfully detects the attack against our Fedora

Core 4 system configured with the default SELinux "targeted" policy operating in "enforcing" mode.

6.3 Implementation

We have implemented a Python module for our property enforcement compiler that generates low-level C code for checking properties like those described in this chapter. The generated code is then linked with a group of runtime libraries for monitor access and response.

As shown in Figure 6.4, the inputs to our compiler are an ASCII specification file, the source/binary type mappings described in Section 5.3, and the types and values of all kernel symbols. As previously mentioned, because we can automatically extract types and symbols from the Linux kernel, the low-level structure definitions may be omitted from the specification file.

The current version of our compiler operates in two phases. The first phase uses a Python GLR parser [78] to parse the specification file. During the parsing phase, symbols and types are resolved and an object-oriented abstract syntax tree (AST) is generated for each section of the specification file. The parser is also responsible for making sure that only quantified or global variables are referenced in each rule and that only declared sets and relations appear in inclusions. Failure to resolve a variable name, type field, symbol, or set/relation name results in a parser error.

The second compiler phase is responsible for generating C code from the AST

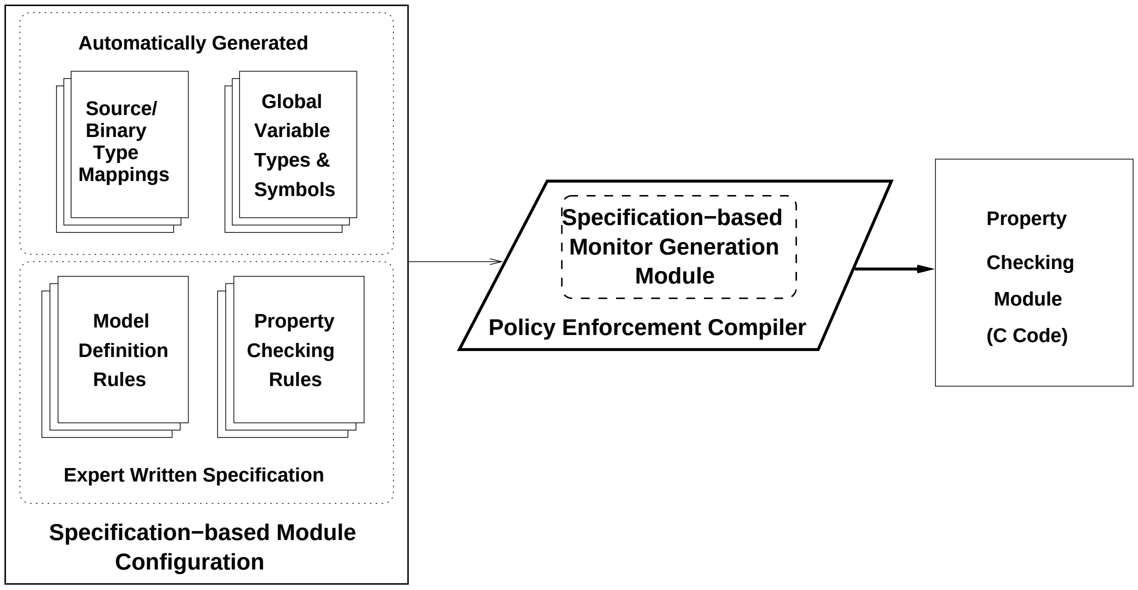
Automatically Generated
Source/ Binary Type Mappings
Global Variable Types & Symbols
Model Definition Rules
Property Checking Rules
Expert Written Specification
Specification−based Module Configuration
Specification−based Monitor Generation Module
Policy Enforcement Compiler
Property Checking Module (C Code)

Figure 6.4: Specification compiler inputs and outputs.

produced during the parsing phase. Our current implementation generates one C function for each model building or property checking rule (hereafter, referred to as "rule functions"). Similarly, each quantifier is converted to a loop and each guard is converted to a conditional. References to global variables or in-memory data structures are translated into their corresponding monitor read operations. Finally, all references to members of declared sets or relations, such as in quantifiers or inclusions, result in a C library call to the corresponding set or relation operation.

Before code generation can begin, the compiler must perform a series of semantic checks. Most notably, rules must be analyzed to identify order dependencies based on sets that are modified (i.e., referenced in the inclusion portion) or read from (i.e., referenced in the quantifiers). At runtime, rule functions are executed in an order consistent with these dependencies such that a given set or relation is completely defined before being quantified as part of another rule. Currently, circular rule dependencies are not supported and a compiler error is generated if a loop is detected.

The code generation phase also performs a number of optimizations aimed at reducing the execution time of the generated code. In our environment, monitor reads are the primary source of latency. Therefore, we include two optimizations for reducing the number of reads performed by the monitor. First, global variables are read only once from the target's memory during each iteration of the generated code's main loop; cached values are used thereafter. Second, variables referenced in each quantifier or guard are analyzed for read/write dependencies so that reads can be performed as part of the outermost loop. Both of these optimizations prevent the same value from being read multiple times.

We have tested our compiler with both example specifications and successfully demonstrated their effectiveness at detecting our sample attacks.

6.4 Discussion

The approach proposed in this chapter is to detect malicious modifications of kernel memory by comparing actual observed kernel state with a hand-generated specification of correct kernel state. As previously mentioned, our specification syntax is based on a language proposed by Demsky and Rinard [25], but adapted for our setting of a property-based external monitor. In their work, Demsky and Rinard introduced a system for automatically repairing data structure errors in programs based on model and constraint specifications [25]. The goal of their system was to produce optimized data structure error detection and repair algorithms [27] that were guaranteed to terminate [26].

As in Demsky and Rinard's work, we have implemented a specification compiler that generates code that builds a model and checks properties, based on the observed runtime state. However, unlike the environment in which Demsky and Rinard's system functioned, the likely response for our system when a constraint fails is not necessarily repair. In fact, there may be reasons *not* to immediately fix the integrity violation, for example to obtain more forensic information without the attacker becoming aware that he or she has been detected. For this reason, we have chosen to include a `response` term in our property checking rules, thereby providing the specification writer with flexibility.

Furthermore, unlike Demsky and Rinard, in our environment we do not have the benefit of executing within the running kernel that we are checking. Particularly in the case of coprocessor-based monitors, memory accesses are not free and pointer values are not local. In our system, every pointer dereference requires read operations by the low-level monitor. For these reasons, optimizing for repair is not the best approach for our environment. Rather, optimizing for efficient object accesses is more appropriate. Our current implementation performs a small number of optimizations, as described in the previous section. However, a more advanced read scheduling algorithm would be more desirable.

Finally, performing checks asynchronously with the running kernel adds some additional challenges that did not exist in Demsky and Rinard's environment. As described previously, we augment our property checking rules with an optional `consistency` parameter to mitigate this problem in our system.

As we have shown, specifications in our system describe possible correct kernel states, not signatures of known attacks. In this way, our approach is a type of specification-based intrusion detection. We do not follow the approach of traditional signature-based virus scanners. We have provided two example specifications for our system and identified the types of modifications that these specifications can detect. While our examples are useful for demonstrating how the proposed system works, they provide little intuition about how specifications would be developed in a real deployment.

Currently, there are two methods for identifying data properties and writing their corresponding specifications: (1) analyzing and abstracting on known threats and (2) deriving data properties and specifications from a high-level English-language security policy. In the analysis of known threats, the goal is to classify the techniques used by adversaries in previous attacks in order to abstract on these methodologies. The result is the identification of a set of data invariants that may be violated by future attacks. Of course, this approach permits the possibility that new attacks may avoid detection by exploiting only those details of the kernel abstracted out of the specification, leading to an interminable "arms race" between attackers and specification-writers. Nevertheless, this approach is still better than traditional signature-based virus scanning, because each specification has the potential to detect an entire class of similar attacks, rather than only a single instance.

It may be possible to avoid such an arms race by using an alternative approach:

deriving specifications from a high-level English-language security policy rather than from an analysis of known attacks. In this approach, an analyst might begin with a policy such as “no runnable processes shall be hidden” or “my reference monitor enforces my particular mandatory access control policy” and then examine the kernel source to determine which data structures have relevant properties and what those properties should be in order for the high-level policy to hold. The analyst’s task is similar to constructing a formal argument for correctness, except that the end result is a configuration for a runtime monitor.

Chapter 7

Related Work

In this chapter, we present two types of related work. First, we introduce a collection of techniques and technologies, both academic and applied, whose aim is similar to ours — to prevent or detect malicious kernel modifications. Then we outline literature from several related fields, including security, reliability, and databases. The members of this second group fall into one of two categories: either they provide complementary approaches to other aspects of the system integrity problem, or they utilize similar methods or techniques to those described in this thesis but apply them to a different problem or domain. For example, work in the areas of fault monitoring and data structure repair utilize similar techniques to achieve reliability, rather than security.

7.1 Related Work: Kernel Integrity

We are not the first to recognize the importance of the OS kernel, or the threat posed by kernel attacks, such as malicious rootkits. In this section, we highlight several proposed approaches to insuring the integrity of OS kernels.

7.1.1 Virtual Machine-Based Approaches

Garfinkel et al. first proposed using a virtual machine monitor to implement a protected system integrity monitor, including invariant-based kernel protection [35]. Their Livewire system is capable of verifying the kernel's text regions, looking for specific types of data attacks that can be detected by querying the system at different levels, and verifying static function pointer tables (e.g., the system call table). Our implementation utilizes a similar VMM-based mechanism, but enforces a far more comprehensive kernel integrity policy. Specifically, our focus on automated control-flow verification provides significantly greater completeness over the kernel checks performed by Livewire.

Grizzard proposes using a VMM to monitor the kernel's execution and validate its control flow [36]. The system works by rewriting the target kernel to trap all dynamic branches into the VMM before they are performed. The control-flow monitor then verifies that the branch is consistent with the kernel's CFG, determined through prior training runs. Though he does not specifically describe CFI, Grizzard's implementation effectively enforces CFI for the OS kernel. The clear advantage is that, by enforcing true CFI, all violations of the CFG, even those that are transient, can be detected. The primary disadvantage of this approach is the incurred overhead and the challenges facing the reduction of that overhead. For the lmbench synthetic benchmark, Grizzard reports an average of 30% performance penalty (worst case 74%) on top of the VMM's existing overhead. Another challenge is how to handle new kernel modules, particularly since the CFG is obtained via training.

Litty and Lie also propose a VMM-based system, called Manitou, for validating the executing code of both user applications and the kernel within a guest VM [61]. The VMM maintains a list of cryptographic hashes of the in-memory representations of application and kernel-level code pages that may be run within the VM. By default, the VMM sets all guest VM pages as non-executable, using hardware support only recently available on the x86. Attempts to execute such pages fault into the VMM, which will set the execute bit and permit execution only if the offending page's hash matches one in its trusted list (further logic is used to prevent subsequent modifications).

Manitou's use of execute bits prevents unauthorized code from ever executing — a stronger guarantee than our SBCFI property. On the other hand, the approach, as proposed, only enforces that valid code pages are executed, not that execution proceeds according to a valid CFG; this is similar to the first validation option presented in Section 5.3.2.4 and will miss at least some "jump-to-libc"-style attacks, even persistent ones. Manitou's reliance on page faulting requires a VMM-based implementation; our approach can be implemented using a PCI card or other external device for greater tamper-resistance of the monitor and backwards compatibility. As a possible inhibitor to practical deployment, Manitou's use of execute bits prevent its operating in an "audit only" mode in which execution may proceed despite monitor warnings. This facilitates a trivial denial of service by an attacker that is able to modify on-disk executables. Finally, overhead is incurred on every change to executable content, such as during normal OS demand-paging. Indeed, the approach has yet to be thoroughly studied on real systems, and so the details of a practical implementation and its overhead are as yet unknown.

In SecVisor, Seshadri et al. utilize new hardware virtualization features in x86-based processors to implement a lightweight hypervisor for guaranteeing operating system code integrity [92]. Similar to Manitou, SecVisor provides guarantees that unauthorized kernel-level (but not user) code is never executed. However, unlike Manitou, SecVisor does not utilize a full-blown VMM, in the hopes of reducing overhead and simplifying verification. As with Manitou, SecVisor is only applicable to VMM-based solutions and imposes higher overhead than is desirable. Future processor support for features such as nested page tables [5] may reduce the overhead imposed due to SecVisor's guarantees; current implementations demonstrate overhead of greater than 10% [92].

7.1.2 SMP-Based Monitors

One of the first proposals for monitoring commodity operating systems was suggested by Hollingworth and Redmond [45], who proposed using one CPU of a symmetric multiprocessor (SMP) machine as an "oversight" processor. In their system, the oversight processor implements security-critical functionality, including autonomously monitoring the memory space of a second processor (the "application" processor) to identify inconsistencies such as anomalous consumption of operating system resources. However, Hollingworth and Redmond provide few specifics about the types of checks they expect to be performed on their system.

Without modifying the underlying hardware or using an additional software layer, such as a VMM, it is unlikely that an SMP-based monitor could be completely effective. First, most SMP machines are designed to provide all processors with full access to RAM, thereby making it difficult to protect the oversight processor's code and data from the application processor. Additionally, oversight processors do not have access to registers or caches present on application processors, thereby facing the same limitations as described in Chapter 4 for our Copilot monitor. Managing resources, such as disks and network cards, is also likely to be problematic. A VMM or small trusted kernel could be used to address all of these problems.

7.1.3 Coprocessor-Based Monitors

Aside from the work of Hollingworth and Redmond [45], who characterized their SMP system in the context of information warfare (IW), Zhang et al. were the first to discuss using an external monitor of system memory for performing intrusion detection [107]. They describe a coprocessor-based approach that utilizes a tamper-resistant device, attached via the PCI bus, that operates in isolation of the target kernel, like our Copilot prototype. While they did not implement their approach, Zhang et al. also described the use of object invariants, such as those used in our specification language module, as targets. Unlike our approach, Zhang et al. proposed using empirical invariants observed through common execution, rather than fundamental properties of the implemented kernel. Additionally, our SBCFI module provides far more complete coverage for control-flow attacks.

7.1.4 Monitors Based on Built-In Hardware Features

Some current and soon-to-be-available hardware features provide built-in functionality that can effectively be used to implement protected execution environments. For example, system management mode (SMM) is a special processor mode on x86 processors (since i386) that provides trusted execution in a separate address space from the kernel or processes [48]. As demonstrated by Heine and Kouskoulas, SMM can be used to develop a monitoring system capable of checking OS kernels without relying upon them [43].

The primary disadvantage of SMM is its limited execution environment. SMM is only entered by means of an external interrupt. Furthermore, updates present a significant challenge and developing a reasonable independent communication mechanism is non-trivial. Normal processor interrupts are also disabled while in SMM, making it difficult to share the processor with a standard OS when extended periods of monitor checking are needed.

More recently, it has been demonstrated that the virtualization extensions available in current processors can be used to develop lightweight hypervisors that provide only security services, rather than full-blown resource virtualization (see SecVisor [92], already discussed). These approaches utilize the built-in protections of hardware virtualization and trusted computing technologies without the administrative and processing costs associated with a full VMM.

Even stronger execution guarantees are provided by Intel's recent Trusted Execution Technology (TXT) and AMD's Secure Virtual Machine (SVM) extensions.

Several features of these technologies enable secure execution. First, both provide forms of memory protection from DMA transfers, thereby protecting sensitive code and data from malicious devices. In addition, systems equipped with a third-party Trusted Platform Module (TPM) can utilize so-called "late launch" features that provide verifiable initial software environments [49, 5]. These technologies represent the most promising platforms for developing trusted software monitoring systems in the near-term.

7.1.5 Software Attestation

Code attestation [55, 34, 94, 89, 90, 96] is a technique by which a remote party, the "challenger" or "verifier," can verify the authenticity of code running on a particular machine, the "attestor." Attestation is typically achieved via a set of measurements performed on the attestor that are subsequently sent to the challenger, who identifies the validity of the measurements as well as the state of the system indicated by those measurements [89]. Both hardware-based [34, 89, 90] and software-based [55, 94] attestation systems have been developed. Measurement typically occurs just before a particular piece of code is loaded, such as between two stages of the boot process, before a kernel loads an new kernel module, or when a kernel loads a program to be executed in userspace [89].

All of the hardware-based systems referenced in this section utilize the Trusted Computing Group's (TCG) [1] Trusted Platform Module (TPM), or a device with similar properties, as a hardware root of trust that validates measurements prior to software being loaded. Software-based attestation systems attempt to provide similar guarantees to those that utilize trusted hardware and typically rely on well-engineered verification functions that, when modified by an attacker, will probabilistically produce incorrect output or take noticeably longer to run. This deviation of output or running time is designed to be significant enough to alert the verifier of foul play.

Traditional attestation systems verify only binary properties of static code and data. In such systems, the only runtime benefit provided is the detection of illegal modifications that utilize well-documented transitions or interfaces where a measurement has explicitly been inserted before the malicious software was loaded. Unfortunately, as reflected in our threat model and assumptions (Chapter 3), attackers are frequently not limited to using only these interfaces.

Haldar et al. have proposed a system known as "semantic remote attestation" [40] in an attempt to extend the types of information the verifying party can learn about the attesting system. Their approach is to use a language-based trusted virtual machine that allows the measurement agent to perform detailed analysis of the application rather than simple binary checksums. The basic principle is that language-based analysis can provide much more semantic information about the properties of an application. Their approach does not extend to semantic properties of the kernel and, because their VM runs on top of a standard kernel, there is a requirement for traditional attestation to bootstrap the system.

Recently, Loscocco et al. introduced "contextual inspection," an approach for performing generic measurement of the Linux kernel's runtime state as part of traditional attestation services [63]. Their implementation, called LKIM, performs object traversals similar to those described in Section 5.3.2 (without the corresponding verification steps) and reports authenticated summaries of the discovered objects to a remote party, who can then verify the kernel's integrity based on a policy of the verifier's choosing. In the context of our architecture, described in Chapter 3, contextual inspection can be thought of as an adaptation whereby the enforcement engine is located on a completely separate machine from the low-level monitor. LKIM focuses on the collection and attestation of kernel objects and does not provide insight into the types of properties that should be verified.

7.1.6 Verifiable Code Execution

Verifiable code execution is a stronger property than attestation whereby a verifier can guarantee that a particular piece of code actually runs on a target platform. This contrasts traditional attestation, where only the loading of a particular piece of software can be guaranteed; once that software is loaded, however, it may be compromised by an advanced adversary. With verifiable code execution, such a modification should not be possible without detection by the verifier. Both hardware-based [17, 96] and, more recently, software-based [93] verifiable code execution systems have been proposed.

In Pioneer [93], Seshadri et al. implement an approach for verifying the execution of arbitrary procedures on untrusted platforms by developing a cleverly-crafted verification function. The verification function computes a checksum over its own code and is designed in such a way that, if it were modified or simulated, the modification would be detected.

In its current form, Pioneer is impractical for real systems because of its strong operational requirements. Specifically, the need to disable interrupts and suspend other execution while using the trusted environment make it comparable to SMM-based monitors. Additionally, its reliance on detailed knowledge of the performance characteristics of the target machine present challenges to general deployment. However, if improvements were made to these requirements, software-based solutions could become a viable approach for building a kernel monitor.

7.1.7 Rootkit Detection

The commercial and applied security communities are also interested in kernel integrity and rootkit detection (examples include Rootkit Profiler [57], RootkitRevealer [20], and Blacklight [31]). Existing tools look for specific signs of compromise within the system by verifying invariants of low-level data structures, by observing system API behavior for inconsistencies, or both. Some tools enforce control-flow properties for a small subset of kernel text and data. As an example of the latter, Microsoft's Patch Guard technology [33, 97] periodically analyzes portions of the Windows kernel, such as the service descriptor table, IDT, and some kernel text, to determine whether those objects have been illegally modified. If a violation is detected, the system is disabled or rebooted and is assumed to be infected.

Our approach can be viewed as a generalization of these techniques that provides far more complete protection. For example, a number of recent tools verify specific kernel function pointers, such as those found in well-known jump tables and kernel subsystems (e.g., the virtual file system) [57]. As a result, improvising attackers have turned to making modifications deeper within the system, finding function pointers and code not verified by current tools [3]. Our SBCFI module removes this avenue from the attacker by checking a much larger, systematically-determined set of function pointers. Additionally, our use of an isolated monitor provides much greater assurance that the implemented checks will be performed, rather than disabled by the attacker.

7.1.8 Static Analysis and Rewriting

As an alternative or complement to dynamic monitoring, the kernel's vulnerability to attack can be reduced by using static analysis and special compilation. Deputy [21, 109] is a compiler and annotation system for C with which programmers can enforce partial memory safety (a garbage collector is required for complete protection), thereby ensuring attackers cannot overrun buffers or perform similar attacks to inject illicit functionality into the kernel in the first place. Related systems include Cyclone [53] and CCured [75], but both of these require representation-modifying compilation (e.g., to introduce so-called "fat" pointers). All three systems have been used to write kernel components [109, 22, 101], but to date none has proven practical for a complete kernel. We observe that some of Deputy's annotations would be useful in our SBCFI module implementation, though we do not require annotations on or within functions, and would not require special compilation.

While the above systems operate at the source level, the program's binary can be rewritten to insert integrity-protecting checks that occur during execution. Abadi et al. implement CFI enforcement in this way [2]. Their implementation is an instance of a general technique called in-line reference monitoring (IRM) that can be used to enforce a range of policies [30, 104]. IRMs typically rely on control-flow and other restrictions to guarantee that their checks are not circumvented (e.g., jumping to code that executes immediately after the check). These restrictions can be difficult to enforce within the kernel, as described in Section 5.1.

7.2 Other Related Work

In this section, we present a collection of works from the related fields of security, reliability, forensics, and databases. We begin by describing complementary approaches to system integrity, including filesystem integrity checkers and trusted boot, which have both inspired the work presented in this dissertation and remain critical for maintaining the integrity of today's systems. Then we introduce a group of works from related problem domains that utilize techniques and strategies similar to ours.

7.2.1 Filesystem Integrity Checkers

The malicious modification of system resources has long been recognized as a dangerous threat in computer systems. One of the most frequent targets of these attacks is the modification of important system files, such as administrator or user applications and their related data and configuration files. Filesystem integrity checkers such as Tripwire are a class of security applications designed to detect the malicious modification of user and system files [56]. These tools, which are intended to be run regularly as part of normal system maintenance, compare the contents and meta-data of system files against a database of "known-good" file attributes. For example, file contents are hashed with a cryptographically secure hash function and compared against a recorded hash value of the original file. Because of the properties of the hash function, any file modification is (with high probability) likely to result in a different hash result. Similarly, file permissions and access times may also be checked for anomalies.

Since the inception of Tripwire and similar tools, a number of application-level integrity checkers have been developed. Similar "binary" or signature-based checks are used, for example, to check for modifications of the Windows Registry database and search for known viruses or malware on the system. While these tools play a useful role in a defense-in-depth strategy, they are not without limitation. First, any file that is updated in a frequent manner, such as user documents or log files, clearly cannot be protected using this strategy. Second, as previously discussed, in order to trust the results of these tools, one must also trust the libraries and kernel filesystem functions to return correct information to the application itself. Molina addresses this issue by performing filesystem checks from a trusted monitor that does not rely on the operating system for filesystem access [70].

7.2.2 Secure Bootstrap

Investigations into secure bootstrap have demonstrated the use of chained integrity checks for verifying the validity of the host kernel [52, 7]. These checks use hashes to verify the integrity of the host kernel and its bootstrap loader at strategic points during the host's bootstrap procedure. At the end of this bootstrap procedure, these checks provide evidence that the host kernel has booted into a desirable state. Our proposed integrity monitor is designed to operate after host kernel bootstrap is complete and provides evidence that the host kernel remains in a desirable state during its runtime.

7.2.3 Coprocessor-Based Security

Many projects have explored the use of coprocessors for security. The same separation from the host that allows the Copilot monitor to operate despite host kernel compromise is also useful for the protection and safe manipulation of cryptographic secrets [106, 39, 51, 60].

Our Copilot monitor prototype is the successor of an earlier filesystem integrity monitor prototype developed by Molina on the EBSA-285 PCI add-in card hardware [69, 71]. Our first version of Copilot was implemented on the same hardware [83]. Rather than examining a host's kernel memory over the PCI bus, Molina's monitor requested blocks from the host's disks and examined the contents of their filesystems for evidence of rootkit modifications.

Closely related to the runtime monitoring of kernel memory is the use of special-purpose hardware to capture relevant forensics state shortly after an attack. Two proposed approaches are Carrier and Grand's Tribble [16], a coprocessor-based forensic memory acquisition tool, and firewire-based memory tools [28]. Both of these approaches represent after-the-fact analysis, rather than runtime monitoring.

7.2.4 Forensic Memory Analysis

To analyze the data collected from hardware devices like Tribble [16] in the wake of an attack, researchers and practitioners have started to develop toolsets for analyzing volatile memory images from potentially compromised machines. The techniques utilized in volatile memory forensic analysis can be similar, if not identical, to the types of checks we perform at runtime. In fact, as we show in FATKit [86], our Python framework can be extended to perform post-mortem analysis when augmented with interactive and visualization usability features.

7.2.5 Specification-Based Intrusion Detection

Specification-based intrusion detection is a technique whereby the detection system's security policy is based on a specification that describes the correct operation of the monitored entity [58]. This approach contrasts signature-based approaches, which look for known threats, and statistical approaches for modeling normalcy in an operational system. Typically, specification-based intrusion detection has been used to describe program behavior, rather than correct state as we have used it [58, 59, 91]. More recently, specifications have been used for network-based intrusion detection as well [102].

7.2.6 Software Fault Monitoring

A runtime software monitor is a system that observes a target program as it runs to determine whether the target's behavior matches a set of predetermined properties [24]. Delgado et al. recently produced a comprehensive survey and taxonomy of existing runtime monitoring systems [24]. While all of the systems considered were primarily focused on fault monitoring, not malicious memory modification, the survey provides excellent insight into the design decisions facing our work. For the sake of consistency, we adopt the terminology proposed by Delgado et al. as follows (quoted from [24]):

- "Software requirements are implementation-independent descriptions of the external behavior of a computation. They answer the question: What behaviors of the software are acceptable?"
- "Software properties are relations within and among states of a computation. Equivalently, software properties can be defined as a set of sequences of states. They answer the question: What relations about states of a computation lead to acceptable external behavior?"

- "A specification language is a language used to specify requirements of the problem domain and other properties associated with software behavior."

Given a program P, the state of a program Σ_P is defined as the accumulated storage (main memory, registers, etc.) of all executing program threads [24]. As the target program executes (i.e., progresses through a sequence of states), a low-level monitor observes changes in the target's state and makes decisions about which states should be analyzed for specific properties, as defined by the monitor's requirements specification. Once a particular state is analyzed, a response such as writing an error to a log file or terminating the program may be initiated.

There are several critical distinctions that separate our work from that of the fault monitoring community. First, our threat model focuses on malicious modification, unlike the bug-related safety concerns of fault monitors. The need to protect the monitor itself from malicious modification places considerable constraints on the mechanisms available to a kernel monitor. For example, many fault monitoring systems rely on instrumented/modified executables that activate the monitor's checks at key points during execution. Such instrumentation could be modified or disabled by an attacker under our threat assumptions.

A second difference between kernel integrity monitoring and fault monitoring is in the types of properties that the monitor may be interested in checking. Many fault monitors make assumptions about the safety of certain parts of the system, such as the immutability of the code itself or of statically-checked constraints. These assumptions are not valid under our threat model. Finally, the vast majority of work on fault monitoring to date has focused on application-level code rather than the kernel. Application monitors benefit from and rely on kernel services that are not available to or are untrusted by a kernel monitor.

7.2.7 Data Structure Consistency and Repair

Demsky and Rinard introduced a system for automatically repairing data structure errors based on model and constraint specifications [25]. The goal of their system was to produce optimized data structure error detection and repair algorithms [27] that were guaranteed to terminate [26]. This work places one level of abstraction on top of the historical 5ESS [42] and MVS [72] work: in those systems, the inconsistency detection and repair procedures were coded manually. By design, Demsky and Rinard's system worked by linking their checking and repair code with the target system. Additionally, the focus of their work was on safety and not security – there is no protection for their monitor code and, rather than alert an administrator of errors, their repair technique allows the program to silently continue without failing.

In similar work, Nentwich and others [76] have developed xlinkit, a tool that detects inconsistencies between distributed versions of collaboratively-developed documents structured in XML [15]. It does so based on consistency constraints written manually in a specification language based on first-order logic and XPath [19] expressions. These constraints deal with XML tags and values, such as "every item in this container should have a unique name value." In later work [77], they describe a tool which analyzes these constraints and generates a set of repair actions. Actions for the above example might include deleting or renaming items with non-unique names. Human intervention is required to prune repair actions from the list and to pick the most appropriate action from the list at repair time.

7.2.8 Database Integrity

There is a long history of concern for the correct and consistent representation of data within databases. Hammer and McLeod addressed the issue in the mid 1970s as it applies to data stored in a relational database [41]. The concept of insuring transactional consistency on modifications to a database is analogous to that of doing resource accounting within the operating system. The database, like the operating system, assumes that data will be modified only by authorized parties through predefined interfaces. While the environments are very different, Hammer and McLeod's work provides excellent insight for us regarding constraint verification. Their system includes a set of constraints over database relations that include an assertion (a predicate), a validity requirement to identify when the constraint should be applied, and a violation action mechanism for updating the database. Hammer and McLeod argue that assertions should not be general purpose predicates (like first-order logic), but should instead be more constrained.

Chapter 8

Conclusions

In this dissertation, we have presented property-based kernel integrity monitoring — a practical and effective approach for detecting real-world attacks against operating system kernels. Our system works by monitoring the kernel's runtime state to check whether it exhibits a set of predetermined security-relevant properties; violations of these properties suggest that the kernel's behavior has been illicitly modified. As we discussed in Chapter 3, the choice of properties is important for the success of our approach. Based on an analysis of all publicly-available Linux rootkits (discussed in Chapter 2), we believe that the most useful properties are those that are both likely to be violated by attackers and practical enough to check. In Chapter 5, we presented one property, which we call state-based control-flow integrity, that meets both of these objectives — all but one of our analyzed rootkits violate SBCFI. For those threats that do not utilize kernel control-flow violations, we provide an effective specification language-based approach for describing other important kernel properties. As experiments performed with our reference implementation show, the two property enforcement systems combine to provide complete detection for all publicly-available Linux kernel rootkits with no false positives in any of our tests.

For security and efficiency reasons, a property-based kernel integrity monitor relies on the use of a trusted mechanism for accessing the kernel's state without depending on its correctness. As the lowest software layer in most systems, the kernel runs at the highest privilege level and has strict performance requirements. In this work, we have presented the design and implementation of two proof-of-concept monitors — a coprocessor-based PCI add-in card, called Copilot, and a virtualization-based solution for the Xen hypervisor. Our VMM-based monitor imposes less than 1% overhead on the target kernel when operating in a reasonable configuration, with no loss of detection coverage.

As our experience and testing show, our approach exhibits all of the characteristics that are critical for its success:

- **Usability:** Our monitor is usable because of its low false positive rate (none experienced in our tests) and simple design. No changes are necessary to the target operating system or any of its libraries or programs. While a small number of annotations of the kernel's types can be useful, no changes to the kernel's source or binary are required. As demonstrated by Copilot, coprocessor-based monitors provide a backwards-compatible solution for the majority of recent x86 computers. Our VMM approach works on the popular Xen hypervisor, which is constantly improving and supports a wide range of modern systems. Only minimal changes (one small patch) were made to Xen for our prototype implementation. Much of our system is automated, with minimal work required by administrators or policy writers. Finally, our modular architecture allows administrators to choose which modules to enable, depending on the characteristics of each, and facilitates the easy integration of future advancements in property enforcement modules.

- **Performance Impact:** Our implemented prototype imposes less than 1% performance penalty on the target system while still meeting effectiveness requirements. Additionally, we have not yet optimized many components of our system. Finally, our system is tunable with regard to performance, allowing administrators to make local decisions and weigh all factors for their individual environments. Specifically, our monitor provides a tuning parameter for spacing its checks over a period of time. This effectively trades off timeliness and, to a lesser degree, completeness for greater performance. The parameter is set at runtime by the system administrator and can be adjusted based on system workload or perceived threat level in deployed environments.

  As described in the previous chapter, the most effective kernel integrity systems proposed to date fall short due to the high overhead that they impose on the protected system. While they may detect many attacks, their operational requirements are too limiting to be used in practice. Property-based integrity monitors strike the necessary balance between effectiveness and efficiency.

- **Detection:** While we have no guarantee that our current implementation will detect all forms of attack, there are several reasons to be optimistic about the long-term viability of property-based integrity monitoring. First, our current implementation is capable of detecting all existing attacks that we know of and likely extensions thereof. Second, our SBCFI property module addresses a broad class of attacks by limiting an attacker's ability to change the kernel's underlying functionality without being detected. This severely limits the range of possibilities available to an attacker who requires stealth.

Our specification-based module is less general, because it requires an expert to describe specific properties. However, this approach is effective in the short term and, more importantly, we believe it is possible to develop more general properties that effectively eliminate entire classes of non-control data attacks. For example, we are currently investigating kernel properties capable of detecting hiding attacks in a general way.

In comparison with current best practice, such as the rootkit detectors described in Section 7.1.7, property-based integrity monitoring is far more general and has the greatest chance of detecting future attacks. Therefore, out of systems that are practical enough to be deployed, our system represents the most effective approach proposed to date.

- **Timeliness:** Both of our implemented property modules are capable of detecting inserted attacks in under three seconds for most configurations. This is much faster than many popular detection systems, such as antivirus scans, and more than sufficient for realistic human response time, the only response mechanism currently implemented by our system.

- **Self-protection:** As we discussed in Chapter 4, our monitor is well-protected from attackers by utilizing a small root of trust, such as a secure coprocessor or trusted hypervisor. While we did not formally verify our implementations, the design of secure coprocessors and hypervisors is an area of ongoing work. Unlike large commodity kernels, which are composed of millions of lines of code, these systems are designed for more effective verification.

Based on our evaluation, we find that our system meets all of its stated goals. We further conclude that property-based monitoring can be an effective and practical approach, and that our implementation represents the best solution for Linux kernel integrity protection proposed to date.

Bibliography

[1] Trusted Computing Group (TCG). http://www.trustedcomputinggroup.org, 2007.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[3] Aditya Kapoor and Ahmed Sallam. Rootkits Part 2: A Technical Primer. http://www.mcafee.com/us/local_content/white_papers/wp_rootkits_0407.pdf, 2007.

[4] Advanced Micro Devices. AMD I/O Virtualization Technology (IOMMU) Specification, February 2007. Order Number: 34434.

[5] Advanced Micro Devices. AMD64 Architecture Programmers Manual Volume 2: System Programming, September 2007. Order Number: 24593.

[6] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.

[7] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[8] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *2007 IEEE Symposium on Security and Privacy*, May 2007.

[9] Eric Bangeman. Gartner: virtualization to rule server room by 2010. http://arstechnica.com/news.ars/post/20070508-gartner-virtualization-to-rule-server-room-by-2010.html, May 2007.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization . In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[11] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, September 1985.

[12] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[13] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote Repair of Operating System State Using Backdoors. In *First International Conference on Autonomic Computing (ICAC)*, pages 256–263, 2004.

[14] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly Media, Inc., third edition, November 2005.

[15] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language. Recommendation REC-xml-20001006, World Wide Web Consortium, October 2000.

[16] Brian D. Carrier and Joe Grand. A Hardware-Based Memory Aquisition Procedure for Digital Investigations. *Journal of Digital Investigations*, 1(1), 2004.

[17] Benjie Chen and Robert Morris. Certifying Program Execution with Secure Processors. In *9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, May 2003.

[18] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the USENIX Security Symposium*, August 2005.

[19] J. Clark and S. Derose. XML Path Language (XPath) Version 1.0. Recommendation REC-xpath-19991116, World Wide Web Consortium, November 1999.

[20] Bryce Cogswell and Mark Russinovich. Microsoft rootkitrevealer. `http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx`, 2007.

[21] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *Proceedings of the European Symposium on Programming (ESOP)*, April 2007.

[22] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the Real World. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[23] Evan Cooke, Farnam Jahanian, and Danny McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop*, pages 39–44, Cambridge, MA, 2005. USENIX Association.

[24] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[25] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003.

[26] Brian Demsky and Martin Rinard. Static Specification Analysis for Termination of Specification-Based Data Structure Repair. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, November 2003.

[27] Brian Demsky and Martin Rinard. Data Structure Repair Using Goal-Directed Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[28] Maximillian Dornseif. 0wn3d by an iPod: Firewire/1394 Issues. In *Proceedings of the 2004 PacSec Applied Security Conference*, 2004.

[29] R. Droms. Dynamic host configuration protocol. Technical Report RFC 2131, Bucknell University, March 1997.

[30] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: a Retrospective. In *Proceedings of the Workshop on New Security Paradigms*, 2000.

[31] F-Secure. F-Secure Blacklight. `http://www.f-secure.com/blacklight/blacklight.html`, 2007.

[32] David Ferraiolo and Richard Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Baltimore, Maryland, October 1992.

[33] Scott Field. Windows Vista Security: An Introduction to Kernel Patch Protection. `http://blogs.msdn.com/windowsvistasecurity/archive/2006/08/11/695993.aspx`, August 2006.

[34] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[35] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, February 2003.

[36] Julian Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

[37] Julian B. Grizzard, John G. Levine, and Henry L. Owen. Re-establishing Trust in Compromised Systems: Recovering from Rootkits That Trojan the System Call Table. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, pages 369–384, Sophia Antipolis, France, September 2004.

[38] PCI Special Interest Group. PCI Local Bus Specification Revision 2.3, March 2002.

[39] Peter Gutmann. An Open-source Cryptographic Coprocessor. In *Proceedings of the 9th USENIX Security Symposium*, pages 97–112, Denver, Colorado, August 2000.

[40] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. In *Proceedings of the 3rd USENIX Virtual Machine Research & Technology Symposium*, May 2004.

[41] Michael Hammer and Dennis McLeod. A Framework For Data Base Semantic Integrity. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, San Francisco, CA, October 1976.

[42] G. Haugk, F.M. Lax, R.D. Royer, and J.R. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT & T Technical Journal*, 64 part 2(6):1385 – 1416, July-August 1985.

[43] D. Heine and Y. Kouskoulas. N-force daemon prototype technical description. Technical Report VS-03-021, The Johns Hopkins University Applied Physics Laboratory, July 2003.

[44] Hewlett-Packard Company. Netperf. `http://www.netperf.org/netperf/`, 2007.

[45] Dennis Hollingworth and Timothy Redmond. Enhancing operating system resistance to information warfare. In *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, volume 2, pages 1037–1041, Los Angeles, CA, USA, October 2000.

[46] IBM Corporation. PowerPC 405GP Embedded Processor Users Manual, August 2003. Order Number: GK10-3118-08.

[47] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, November 2007. Order Number: 253668-025US.

[48] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2, November 2007. Order Number: 253669-025US.

[49] Intel Corporation. Intel Trusted Execution Technology Preliminary Architecture Specification, August 2007. Order Number: 31516804.

[50] Intel Corporation. Intel Virtualization Technology for Directed I/O Architecture Specification, September 2007. Order Number: D51397-003.

[51] Naomaru Itoi. Secure Coprocessor Integration with Kerberos V5. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[52] Naomaru Itoi, William A. Arbaugh, Samuela J. Pollack, and Daniel M. Reeves. Personal Secure Booting. *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, 2119:130–144, February 2001.

[53] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[54] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs with Type Inference. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[55] Rick Kennell and Leah H. Jamieson. Estabilishing the Genuity of Remote Computer Systems. In *Proceedings of the USENIX Security Symposium*, August 2003.

[56] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)*, pages 18–29, Fairfax, Virgina, November 1994.

[57] Tobias Klein. Rootkit profiler LX. `http://www.trapkit.de/research/rkprofiler/rkplx/rkplx.html`, 2007.

[58] Calvin Ko, George Fink, and Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, 1994.

[59] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[60] Mark Lindemann and Sean W. Smith. Improving DES Coprocessor Throughput for Short Operations. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[61] Lionel Litty and David Lie. Manitou: a layer-below approach to fighting malware. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, 2006.

[62] Peter A. Loscocco and Stephen D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[63] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux Kernel Integrity Measurement Using Contextual Inspection. In *STC '07: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 21–29, Alexandria, Virginia, USA, 2007. ACM.

[64] Robert Love. *Linux Kernel Development*. Novell Press, second edition, 2005.

[65] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[66] Laurianne McLaughlin. Bot software spreads, causes new worries. *IEEE Distributed Systems Online*, 5(6):1, 2004.

[67] Matt Miller and Jarkko Turkulainen. Remote Library Injection. `http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf`, April 2004.

[68] P. Mockapetris. Domain names—concepts and facilities. Technical Report RFC 1034, ISI, November 1987.

[69] Jesus Molina. Using Independent Auditors for Intrusion Detection Systems. Master's thesis, University of Maryland at College Park, 2001.

[70] Jesus Molina and William A. Arbaugh. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, pages 291–302, Singapore, December 2002.

[71] Jesus Molina and William A. Arbaugh. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, pages 291–302, Singapore, December 2002.

[72] Samiha Mourad and Dorothy Andrews. On the Reliability of the IBM MVS/XA Operating System. *IEEE Transactions on Software Engineering*, 13(10):1135–1139, 1987.

[73] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Cr iteria*, December 1985.

[74] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, April 2002.

[75] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[76] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151 – 185, May 2002.

[77] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering*, May 2003.

[78] Tim Newsham. PyGgy. `http://www.lava.net/~newsham/pyggy/`, October 2004.

[79] Federal Bureau of Investigation. 2005 FBI Computer Crime Survey. `http://www.fbi.gov/publications/ccs2005.pdf`, January 2006.

[80] Federal Bureau of Investigation. Over 1 Million Potential Victims of Botnet Cyber Crime. `http://www.fbi.gov/pressrel/pressrel07/botnet061307.htm`, June 2007.

[81] David A. Patterson and John L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann Publishers, second edition, 1998.

[82] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[83] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the USENIX Security Symposium*, August 2004.

[84] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the USENIX Security Symposium*, August 2006.

[85] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, October 2007.

[86] Nick L. Petroni, Jr., AAron Walters, Timothy Fraser, and William A. Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4), December 2006.

[87] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *9th USENIX Security Symposium*, Denver, CO, August 2000.

[88] Joanna Rutkowska. Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case). In *Black Hat DC 2007 Briefings*, Arlington, VA, February 2007.

[89] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based Policy Enforcement for Remote Access. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2004.

[90] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the USENIX Security Symposium*, August 2004.

[91] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *8th USENIX Security Symposium*, pages 63–78, Washington, D.C., August 1999.

[92] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, Stevenson, Washington, USA, 2007. ACM.

[93] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[94] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[95] T. Shanley and Don Anderson. *PCI System Architecture*. Addison Wesley, fourth edition, 1999.

[96] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[97] skape and Skywing. Bypassing PatchGuard on Windows x64. *Uninformed*, Volume 3, January 2006. `http://www.uninformed.org/?v=3`.

[98] Solar Designer. Getting around non-executable stack (and fix). Bugtraq, August 1997.

[99] SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org`, 2007.

[100] Joe Stewart. Storm worm ddos attack. `http://www.secureworks.com/research/threats/storm-worm`, February 2007.

[101] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Science of Computer Programming (SCP)*, 62(2):122–144, October 2006. Special issue on memory management.

[102] C.Y. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. Levitt. A Specification-Based Instrusion Detection System for AODV. In *2003 ACM Workshop on security of Ad Hoc and Sensor Networks (SASN '03)*, Fairfax, VA, October 2003.

[103] VMware, Inc. VMware Workstation. `http://www.vmware.com`, 2007.

[104] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

[105] XenSource, Inc. A Performance Comparison of Commercial Hypervisors. `http://www.xensource.com/files/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf`, 2007.

[106] Bennet Yee and J. D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 155–170, New York, New York, July 1995.

[107] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the ACM SIGOPS European Workshop*, September 2002.

[108] Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *9th USENIX Security Symposium*, Denver, CO, August 2000.

[109] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, November 2006.