Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor

Nick L. Petroni, Jr.
npetroni@cs.umd.edu
Department of Computer Science

Timothy Fraser
tfraser@umiacs.umd.edu
Institute for Advanced Computer Studies

Jesus Molina
chus@cs.umd.edu
Department of Computer Science

William A. Arbaugh
waa@cs.umd.edu
Department of Computer Science

University of Maryland, College Park, MD 20742, USA

Abstract

Copilot is a coprocessor-based kernel integrity monitor for commodity systems. Copilot is designed to detect malicious modifications to a host's kernel and has correctly detected the presence of 12 real-world rootkits, each within 30 seconds of their installation with less than a 1% penalty to the host's performance. Copilot requires no modifications to the protected host's software and can be expected to operate correctly even when the host kernel is thoroughly compromised – an advantage over traditional monitors designed to run on the host itself.

1 Introduction

One of the fundamental goals of computer security is to ensure the integrity of system resources. Because all user applications rely on the integrity of the kernel and core system utilities, the compromise of any one part of the system can result in a complete lack of reliability in the system as a whole. Particularly in the case of commodity operating systems, the ability to place assurance on the numerous and complex parts of the system is exceedingly difficult. The most important pieces of this complex system reside in the core of the kernel itself. While a variety of tools and architectures have been developed for the protection of kernel integrity on commodity systems, most have a fundamental flaw – they rely on some portion of kernel correctness to remain trustworthy themselves. In a world of increasingly sophisticated attackers, this assumption is frequently invalid. To address the growing need for integrity protection that does not rely on kernel correctness, we designed Copilot – a kernel integrity monitor that does not rely on the kernel for access to main memory and requires no modifications to the protected host's software.

The ability to perform arbitrary checks on system memory provides Copilot with a mechanism to monitor for any number of signs that the kernel is no longer operating in a trustworthy manner. To exemplify the usefulness of this approach, we present a prototype based on a PCI add-in card that detects malicious modifications to Linux kernels. This prototype extends previous work on auditing filesystem changes [23, 24] and has been shown to successfully perform its audit of system memory every 30 seconds with less than a 1% penalty to system performance. This efficiency, combined with the added assurance provided by the Copilot architecture, results in a considerable advancement in the protection of commodity operating system kernels running on commodity hardware. In addition to the detection possibilities demonstrated by this prototype, the Copilot architecture provides for future advancements including partial restoration of changes due to malicious modifications and a more general scheme for configuration management.

As an example of Copilot's integrity monitoring technique, we have tested and verified the prototype's ability to successfully detect the presence of twelve commonly-used, publicly-known rootkits. Rootkits are collections of programs that enable attackers who have gained administrative control of a host to modify that host's software, usually causing it to hide their presence from the host's genuine administrators. Every popular rootkit soon encourages the development of a program designed to detect it; every new detection program inspires rootkit authors to find better ways to hide. But in this race, the rootkit designers have traditionally held the advantage: the most sophisticated rootkits modify the operating system kernel of the compromised host to secretly work on behalf of the attacker. When an attacker can arbitrarily change the functionality of the ker-

This project was sponsored in part by the National Science Foundation (ANI0133092), the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0535. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the offi cial policies or endorsements, either expressed or implied of the U.S. Government.

nel, no user program that runs on the system can be trusted to produce correct results – including user programs for detecting rootkits.

The Copilot monitor prototype is designed to monitor the 2.4 and 2.6 series of Linux kernels. Copilot's aim is not to harden these kernels against compromise, but to detect cases where an attacker applies a rootkit to an already-compromised host kernel. Copilot is designed to be effective regardless of the reason for the initial compromise – be it a software or configuration flaw or a human error involving a stolen administrative password.

The remainder of this section provides an overview of the Copilot monitor prototype testbed. Section 2 contains a brief survey of some existing kernel-modifying rootkits and their behaviors, followed by a complimentary survey of existing rootkit detection software in Section 3. Sections 4 through 6 discuss the implementation of the prototype. The Copilot monitor depends upon a number of specific features of the IBM PC-compatible PCI bus (Section 4) and the Linux virtual memory subsystem (Section 5) in order to operate. Section 6 describes how the Copilot monitor uses these features to provide useful integrity monitoring functionality.

Depending on the aggressiveness of its configuration, the Copilot monitor's examination of host RAM can lead to some contention on the PCI bus and for access to main memory. Section 7 presents the results of several benchmarks examining the trade-off between reducing the average amount of time required by Copilot to detect a rootkit and increasing the amount of PCI bus capacity Copilot leaves for the host's own applications.

Section 8 discusses the limitations of the present Copilot monitor prototype, and Section 9 discusses possible avenues of future work. Section 10 compares the Copilot monitor to other related work, and Section 11 presents our conclusions.

The Copilot monitor prototype testbed consists of two machines and a PCI add-in card, shown in figure 1. On the left is the *host* – the machine that Copilot monitors for the presence of rootkits. It contains the Copilot *monitor* on its PCI add-in card. On the right is the *admin station* – the machine from which an administrator can interact with the Copilot monitor. The remainder of this document deliberately uses the words host, monitor, and admin station to distinguish between these three entities. The phrase "host kernel," in particular, always refers to the kernel running on the host – the machine that Copilot monitors. All machines run versions of the GNU/Linux operating system; their configurations are described more fully in Section 7.

The host is a desktop PC configured as a server. The monitor is an Intel StrongARM EBSA-285 Evaluation Board - a single-board computer on a PCI add-in card inserted into the host's PCI bus. The monitor retrieves parts of host RAM for examination through Direct Memory Access (DMA) without the knowledge or intervention of the host kernel.

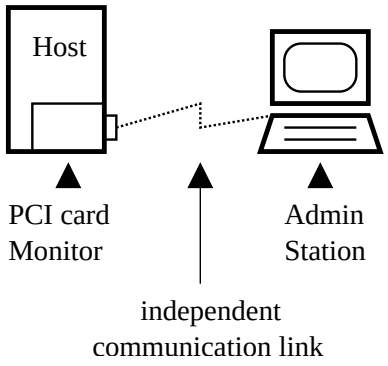

Figure 1: Copilot monitor prototype testbed architecture

The admin station is a laptop workstation that connects to the monitor via an independent communication link attached to a connector on the back of the EBSA-285 Evaluation Board. This independent link allows the monitor to send reports to the admin station without relying upon the (possibly compromised) host for communication. It also allows the admin station to periodically poll the monitor in order to detect cases where the host has been compromised and powered-down, or trapped in a cycle of PCI resets that would prevent all of its PCI devices (the monitor included) from performing useful work.

In an alternate design, the monitor and admin station might communicate via cryptographically-protected messages passed to the host kernel via the PCI bus. This configuration would eliminate the monitor's need for independent communication link hardware, permitting its implementation on a less expensive board. However, the present Copilot monitor prototype uses an independent communication link for two reasons. First, one of the primary goals of Copilot is to protect an unmodified commodity system. Communication over the PCI bus would require a driver on the protected system, thereby violating this goal. Second, Copilot is much more likely to provide system administrators with useful information in the event of an intrusion if its messages are guaranteed to reach the admin station. In the case of a PCI-based reporting mechanism, an attacker could trivially disable reporting in the protected host, forcing the only sign of trouble at the admin station to be Copilot's failure to report.

Presently, the Copilot monitor prototype implements a detection strategy based on MD5 hashes [29] of the host kernel's text, the text of any loaded LKMs, and the contents of some of the host kernel's critical data structures. Like a number of the user-mode detection programs described in Section 3, it calculates "known good" hashes for these items when they are believed to be in a correct, non-

compromised state. The Copilot monitor then periodically recalculates these hashes throughout host kernel run-time and watches for results that differ from the known good values. The Copilot monitor sends reports of differing hash values to the admin station where they can be judged to be the result of either valid administrative kernel modification (such as dynamic driver loading) or of a malicious rootkit. This judgment is made manually on the current prototype; as described in Section 9, automating it is a subject for future work.

2 Rootkit taxonomy

Rootkits are collections of software that enable attackers who have gained administrative control of a host to hide their presence from the host's genuine administrators. Once installed, rootkits modify the host's software to provide an attacker with the ability to hide the existence of chosen processes, files, and network connections from other users. Rootkits may also provide convenient backdoors through which an attacker can regain privileged access to the host at will or keystroke logging facilities for spying on legitimate users. By installing rootkits, attackers increase the ease with which they can return to and exploit a compromised host over the course of weeks or months without being detected.

Rootkits can be partitioned into two classes: those that modify the host operating system kernel and those that do not. Those in the second class are simpler and easier to detect. These simple rootkits replace critical system utilities such as `ps`, `ls`, and `netstat` with modified versions that deceptively fail to report the presence of an attacker's processes, files, and network connections. However, since the operating system kernel is not part of the deception, a suspicious administrator can detect the presence of these simple rootkits by comparing the output of the modified utilities against the output of unmodified versions obtained from read-only installation media, or against information provided directly by the operating system kernel via its `/proc` filesystem [3]. Additionally, defensive software is available which monitors the files containing a host's critical system utilities for the kinds of unexpected changes caused by a simple rootkit installation [18, 24].

Copilot is designed to detect the more complex class of rootkits that modify the host operating system kernel itself to provide deceptive information. The above detection techniques will fail when run on a sufficiently modified kernel: when the kernel itself lies, even correct system utilities can do nothing but pass this false information on to the user. Section 3 describes the race that is in progress between authors of complex rootkit detection tools that depend on at least *some* part of the kernel remaining unmodified and authors of rootkits who respond by increasing the scope of their modifications. The Copilot project demonstrates a means of escaping this race by running integrity monitoring software on a PCI card whose software does not depend on the health of the host operating system kernel being monitored and is beyond the reach of an attacker. Copilot's integrity monitoring strategy is described fully in Section 6.

There are many rootkits designed to modify the Linux kernel available on the Internet. The remainder of this section describes the workings of a representative sample of them, listed in table 1. The present Copilot monitor prototype has successfully detected the presence of each of the rootkits listed in this table within 30 seconds of their being installed on the testbed host.

The example rootkits in table 1 provide a variety of services. Nearly all are designed to make the kernel return incorrect or incomplete information when queried by user-mode programs, in order to hide the presence of the rootkit and the attacker's processes and files. Some of them also provide backdoors allowing remote access to the compromised system, or provide a means of privilege escalation to local users. Some of the example rootkits provide keystroke loggers.

The rootkits in the *complete rootkits* section of table 1 provide a sufficient amount of deceptive functionality that they might be of use to an actual attacker. The remaining rootkits provide only limited functionality and serve only to demonstrate how a particular aspect of rootkit functionality might be implemented.

The check-boxes in the table call attention to the rootkit attributes that are most relevant to a detection tool like Copilot: the means by which attackers load the rootkits into the kernel and the means by which the rootkits cause the kernel to execute their functions.

The first column of table 1 shows that all but one of the example rootkits are implemented as LKMs and are designed to be loaded through the kernel's LKM-loading interface as if they were device drivers. This fact is significant because an unmodified kernel will report the presence of all loaded LKMs – a stealthy rootkit must take pains to modify the kernel or its LKM management data structures to avoid being revealed in these reports. The LKM-loading interface is not the only means of loading a rootkit into the kernel, however. The SucKIT rootkit is designed to be written into kernel memory via the kernel's `/dev/kmem` interface using a user-mode loading program provided with the rootkit. (The `/dev/kmem` interface provides privileged processes with direct access to kernel memory as if it were a file.) This loading method does not use the kernel's LKM-loading interface and consequently leaves no trace in its data structures.

The remaining columns of table 1 show that the example rootkits use a variety of means to cause the kernel to execute their code. Nearly all of them overwrite the addresses of some of the kernel's system call handling functions in

rootkit name:	loads via:	overwrites syscall jump	adds new syscall jump	modifies kernel text	adds hook to /proc	adds inet protocol
Complete rootkits:						
adore 0.42	LKM	x				
knark 2.4.3	LKM	x			x	x
rial	LKM	x				
rkit 1.01	LKM	x				
SucKIT 1.3b	kmem	x	x			
synapsys 0.4	LKM	x				
Demonstrates module or process hiding only:						
modhide1	LKM	x				
phantasmagoria	LKM			x		
phide	LKM	x				
Demonstrates privilege escalation backdoor only:						
kbd 3.0	LKM	x				
taskigt	LKM				x	
Demonstrates key logging only:						
Linspy v2beta2	LKM	x				

Table 1: Features of example Linux kernel-modifying rootkits

the system call table with the addresses of their own doctored system call handling functions. This act of system call interposition causes the kernel to execute the rootkit's doctored system call handling functions rather than its own when a user program makes a system call [11, 9].

The rootkit's doctored functions may implement deceptive functionality in addition to the service normally provided by the system call. For example, rootkits often interpose on the `fork` system call so that they may modify the kernel's process table data structure in a manner which prevents an attacker's processes from appearing in the user-visible process list whenever a the kernel creates a new process. Privilege-escalating backdoors are also common: the rkit rootkit's doctored `setuid` function resets the user and group identity of processes owned by an unprivileged attacker to those of the maximally-privileged `root` user.

System call interposition is not the only means by which rootkits cause the kernel to execute their functions, however. In addition to interposing on existing system calls, the SucKIT rootkit adds new system calls into previously empty slots in the kernel's system call table. The phantasmagoria rootkit avoids the system call table altogether and modifies the machine instructions at the beginnings of several kernel functions to include jumps to its own functions. The knark and taskigt rootkits add hooks to the `/proc` filesystem that cause their functions to be executed when a user program reads from certain `/proc` entries. The taskigt rootkit, for example, provides a hook that grants the `root` user and group identity to any process that reads a particular `/proc` entry. The knark rootkit also registers its own inet protocol handler that causes the kernel to create a privileged process running an arbitrary program when the kernel receives certain kinds of network packets.

3 Existing detection software

A number of tools designed to detect kernel-modifying rootkits are currently available to system administrators. These software packages make a series of checks on any number of system resources to determine if that system is in an anomalous state. In this section, we describe some of the common and novel approaches taken by a subset of kernel-modifying rootkit detectors.

There are two categories of kernel-modifying rootkit detectors: those that check for rootkit symptoms by looking for discrepancies that are detectable from user-space and those that analyze kernel memory directly to detect changes or inconsistencies in kernel text and/or data structures. We refer to these two types of tools as *user-space* and *kernel memory* tools respectively. A number of tools can be considered both user-space and kernel memory tools, as they provide detection mechanisms that fall into both categories. Table 2 summarizes a representative sample of commonly used rootkit detectors that can, at least to some degree, detect kernel-modifying rootkits. Those tools with a mark present in the rightmost column perform user-space checks. Those with marks in either of the two leftmost columns analyze kernel memory through the specified mechanisms.

Even many kernel-modifying rootkits have symptoms that are readily-observable from user-space, without accessing kernel memory or data structures directly. For example, as previously mentioned, some rootkits add entries to the kernel's `/proc` filesystem. Such entries can often

rootkit detector:	Kernel memory access		synchronous detection	user-space symptom detection
	/dev/kmem	detector LKM		
KSTAT	x	x		x
St. Michael		x	x	
Carbonite		x		
Samhain	x			x
chkrootkit				x
checkps				x
Rkscan				x
RootCheck				x
Rootkit Hunter				x

Table 2: kernel-modifying rootkit detector mechanisms

be found with standard directory lookups and, many times, even with trusted, non-compromised versions of `ls`. Similarly, a number of LKM rootkits do a poor job of hiding themselves from simple user queries such as checking for exported symbols in `/proc/ksyms`. These symbols are part of the rootkit's added kernel text and do not exist in healthy kernels.

User-space checks fall into two categories: those that are rootkit-specific and those that are not. The former are extremely efficient at detecting well-known rootkits using simple checks for specific directories, files, kernel symbols, or other attributes of the system. One of the most common rootkit-specific detectors, chkrootkit, has a set of predetermined tests it performs looking for these attributes. In doing so, it can detect dozens of LKM rootkits currently in common use.

Non-rootkit specific checks by user-space tools generally perform two types of tasks. The first is a simple comparison between information provided through the `/proc` filesystem and the same information as determined by system calls or system utilities. One such common check is for process directory entries hidden from `ps` and the `readdir` system call. The second common user-space check is for anomalies in the Linux virtual filesystem directory structure. Some rootkits hide directories, resulting in potential discrepancies between parent-directory link counts and the number of actual subdirectories visible by user programs.

While user-space checks can prove useful under certain conditions, they have two fundamental limitations. First, because they are dependent on interfaces provided by the kernel, even the most critical of compromises can be concealed with an advanced kernel-resident rootkit. Second, most of the symptoms that are detectable from user-space are not general enough to protect against new and unknown rootkits. However, there is a set of tools whose purpose is to protect the kernel in a more general way, by watching for rootkits at the point of attack – in kernel memory. We first describe the mechanisms used by these tools to access kernel memory and the shortcomings with each approach. Then, we provide some general insight into the types of checks kernel memory protectors perform. Finally, we briefly introduce four common tools currently used to detect rootkits using kernel memory.

Not surprisingly, the methods available to rootkit detectors are not unlike those utilized by rootkits themselves. Unfortunately, easy access to kernel memory is a double-edged sword. Although it provides for the convenient extensibility of the Linux kernel through kernel modules, it also provides for the trivial insertion of new kernel code by attackers who have gained root privileges. There are two primary access paths to the Linux kernel, both of which were discussed in Section 2. The first, `/dev/kmem`, allows attackers and protectors alike to write user programs that can arbitrarily change kernel virtual memory. There is much more overhead involved with a program that uses `/dev/kmem`, because symbols need to be ascertained independently (typically from `/proc/ksyms` or the System.map file) and data structures need to be processed manually. However, the portability of a tool written in this way would allow it to work even on kernels built without LKM support. One major drawback which must be considered by authors of tools that use `/dev/kmem` is that the interface is provided by the kernel – the entity whose integrity they seek to verify. Because the interface is provided by a kernel code, there is always potential that a rootkit is providing false information to the user-space tool.

The second method, insertion of an LKM by the tool, can be a far more powerful approach. First, it gives the tool the ability to execute code *in the kernel*, the privileges of which include the ability to manipulate the scheduler, utilize kernel functions, provide additional interfaces to user-space programs, and have immediate access to kernel symbols. The negatives of using an LKM are twofold. First, the approach clearly will not work in a kernel built without LKM support. Second, a rootkit already resident in the kernel could modify, replace, or ignore a module as it sees fit, depending on its sophistication.

Functionality	KSTAT	St. Michael	Carbonite	Samhain
Long-term change detection				
Hidden LKM detection	x	x		
Syscall table change detection	x	x		x
Syscall function change detection	x	x		x
Kernel text modification detection		x		
IDT change detection	x			x
Short-term system state				
Hidden process detection	x		x	
Hidden socket detection	x		x	
Extra features				
Hides self from rootkits		x		x
Restore modified text changes		x		

Table 3: kernel-modifying rootkit detector functionality- detectors that examine kernel memory

Once provided access to kernel memory, tools take a number of approaches to protect the kernel. First, and perhaps the most well-known, is protection of the system call table [16]. As shown in table 1, the vast majority of rootkits utilize system call interposition in one form or another. Rootkit detectors with access to system memory can perform a number of checks on the system call table, the most notable of which is storing a copy or partial copy of the table and the functions to which it points. This copy is then used at a later time to make periodic checks of the table. A similar procedure is also used by some kernel memory-based detectors to check the interrupt descriptor table (IDT), and in one case, the entire kernel text.

In addition to protecting text and jump tables within the kernel, detection tools are used to provide information about kernel data that cannot easily be obtained from user-space. Some common examples are the data structures associated with LKMs, files, sockets, and processes, each of which can change frequently in a running kernel. Tools with access to kernel memory can parse and analyze this data in order to look for suspicious or anomalous instances of these objects. User-space tools that use `/dev/kmem` and LKMs that create new interfaces can compare data obtained directly from the kernel in order to find hidden processes, files, sockets, or LKMs.

Table 3 provides a list of four common kernel memory-based rootkit detection tools. The table also shows a set of functionality that is common among such detectors, as well as the specific functionality provided by each tool. We briefly describe each of these four tools.

KSTAT is a tool for system administrators, used to detect changes to the interrupt descriptor table, system call vector, system call functions, common networking functions, and `proc` filesystem functions. Additionally, it provides an interface for obtaining information about open sockets, loaded kernel modules, and running processes directly from kernel memory. KSTAT relies on `/dev/kmem` for its checking, but uses LKMs in two ways. First, the initial run of KSTAT on a "clean" kernel uses a module to obtain kernel virtual memory addresses for some of the networking and filesystem functions it protects. Second, because the module list head pointer is not exported by the Linux kernel, a "null" module is used to locate the beginning of the module linked list at each check for new modules. Change detection is performed by using "fingerprints" of the original versions. In the case of function protection, this amounts to the copying of a user-defined number of bytes at the beginning of the function. Jump tables (e.g. IDT and system call) are copied in full.

Another popular tool that uses `/dev/kmem` for kernel integrity protection is Samhain, a host-based intrusion detection system (IDS) for Unix/Linux [5]. While rootkit detection is not the only function of Samhain, the tool provides IDT, system call table, and system call function protection similar to that of KSTAT. Although it does not perform all of the functionality with regard to kernel state provided by KSTAT, Samhain does have one additional feature – the ability to hide itself. An LKM can be loaded to hide process and file information for Samhain so that an attacker might not notice the tool's existence when preparing to install a rootkit. Because of this feature, administrators can prevent attackers with root access from recognizing and killing the Samhain process.

The third tool we discuss is likely the most well-known rootkit detector tool available – St. Michael [5, 16]. As part of the St. Jude kernel IDS system, St. Michael attempts to protect kernel text and data from within the kernel itself via an LKM. St. Michael provides most of the same protection as KSTAT and Samhain with a number of added features. First, it replaces copy fingerprints with MD5 or SHA-1 hashes of kernel text and data structures, thereby covering larger segments of that information. Second, St. Michael is the only tool discussed that provides both preventative and reactive measures for kernel modifi-

cations, in addition to its detection features. The former are provided through changes such as turning off write privileges to `/dev/kmem` and performing system call interposition on kernel module functions in order to synchronously monitor kernel changes for damage. Because of its synchronous nature, St. Michael has a distinct advantage in detection time – to the point that it can actually prevent changes in some cases. The final major advantage of the St. Michael system is its ability to restore certain parts of kernel memory in the case of a detected change. By backing up copies of kernel text, St. Michael provides an opportunity to replace modified code before an attacker utilizes changes made by a rootkit. However, St. Michael has the same disadvantages as any LKM-based tool, as described previously.

The final tool we discuss in this section is another LKM-based memory tool, known as Carbonite [16]. While the latest release only works with version 2.2 Linux kernels, the implementation is an excellent example of integrity protection on kernel data structures. Carbonite traces all tasks in the kernel's task list and outputs diagnostic information such as open files, open sockets, environment variables, and arguments. An administrator can then manually audit the output file in search of anomalous entries. Carbonite is a good example of a tool that can be used to produce more specific information after an initial indication of intrusion.

4 Coprocessor Requirements

In order to perform its task of monitoring host memory, the Copilot coprocessor must meet, at a minimum, the following set of requirements:

- *Unrestricted memory access.* The coprocessor must be able to access the system's main memory. Furthermore, it must be able to access the full range of physical memory- a subset is not sufficient.

- *Transparency.* To the maximum degree possible, the coprocessor should not be visible to the host processor. At a minimum, it should not disrupt the host's normal activities and should require no changes to the host's operating system or system software.

- *Independence.* The coprocessor should not rely on the host processor for access to resources – including main memory, logging, address translation, or any other task. The coprocessor must continue to function regardless of the running state of the host machine, particularly when it has been compromised.

- *Sufficient processing power.* The coprocessor will, at a minimum, need to be able to process large amounts of memory efficiently. Additionally, the choice of hashing as a means of integrity protection places on the coprocessor the additional requirement of being able to perform and compare such hashes.

- *Sufficient memory resources.* The coprocessor must contain enough long-term storage to keep a baseline of system state. This summary of a non-compromised host is fundamental to the proper functioning of the auditor. Furthermore, the coprocessor must have sufficient on-board, non-system RAM that can be used for its own private calculations.

- *Out-of-band reporting.* The coprocessor must be able to securely report the state of the host system. To do so, there must be no reliance on a possibly-compromised host, even to perform basic disk or network operations. The coprocessor must have its own secure channel to the admin station.

The EBSA-285 PCI add-in card meets all of the above requirements and provides a strong foundation for an initial prototype. The remainder of this section briefly describes how these requirements are met by our EBSA implementation, including some of the technical details that enable it to work in an i386 host.

4.1 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, the need for direct access to system memory without processor intervention became apparent [26]. 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 [26].

On any given PCI bus, there are two types of devices: initiators, or bus masters, and targets [35]. 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. The EBSA-285 has full bus master functionality, as well as support for DMA communication with host memory [13].

DMA was designed to increase system performance by reducing the amount of processor intervention necessary

for device access to main memory [26]. However, since the ultimate goal is to facilitate communication between the device and the processor, some information must be shared by both parties to determine where in memory information is being stored. In order to account for the separation of address spaces between the bus and main memory, the host processor will typically calculate the translation and notify the device directly where in the PCI address space it should access [26]. Unfortunately for the EBSA, and for our goal of monitoring host memory, this separation makes it difficult to determine where in main memory the device is actually reading or writing; there is not necessarily an easy mapping between PCI memory and system memory. 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 processor.

4.2 Transparency and Independence

As previously mentioned, there are two modes in which the monitor prototype can be run. While in the first mode the monitor loads its initial system image from the running host, the second is a standalone mode that provides for complete independence with regard to process execution [23, 24]. As with all add-in cards, the EBSA remains able to be queried by the host and reliant on the PCI bus for power in both modes of operation. However, in standalone mode, the EBSA can be configured to deny all configuration reads and writes from the host processor, thereby making its execution path immutable by an attacker on the host.

A creative attacker may find ways to disable the device, the most notable of which is sending a PCI-reset to prevent the board from accessing main memory. 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 or resetting all devices on the bus. Rebooting the host serves to bring unnecessary attention to the machine and may not be advantageous to the attacker. Furthermore, if the attacker is connected using a PCI-based network card the PCI reset will also disrupt the attack itself. Second, in a proper configuration, the admin station is a completely separate machine from the monitor and the host. A simple watchdog is placed on the admin station to insure that the monitor reports as expected. If no report is provided after a configurable amount of time, an administrator can easily be notified.

symbol	use
`_text`	beginning of kernel text
`_etext`	end of kernel text
`sys_call_table`	kernel's system call table
`swapper_pg_dir`	kernel's Page Global Directory
`idt_table`	kernel's Interrupt Descriptor Table
`modules`	head of kernel's LKM list

Table 4: Symbols taken from System.map

4.3 Resources

As shown by our implementation, the EBSA has sufficient resources to carryout the necessary operations on host memory. While we have not implemented all possible memory checks, the process of reading and hashing continuously has been tested, and the board has proven to perform reliably. More about these tests can be found in Section 7. In addition to its memory resources, the EBSA also provides a serial (RS-232) connection for external logging and console access by the management station. The board therefore meets all of our requirements.

5 Linux virtual memory

This section describes the two features of the Linux kernel that enable the Copilot monitor to locate specific data structures and regions of text within the host kernel's address space: linear-mapped kernel virtual addresses and the absence of paging in kernel memory. In its present form, Copilot would be unable to effectively monitor a host running a kernel without these features.

The linear-mapped kernel virtual address feature enables the Copilot monitor to locate the regions it must monitor within host kernel memory. As described in Section 1, when the Copilot monitor wishes to examine a region of host kernel memory, it makes a DMA request for that region over the PCI bus. Furthermore, Section 4 explained that, because of the nature of the PCI bus on the PC platform, the Copilot monitor must specify the address of the region to retrieve in terms of the host's physical address space. This requirement is somewhat inconvenient, because Copilot takes the addresses of several interesting symbols from the host kernel or its `System.map` file at Copilot configuration time. (These symbols are listed in table 4.) These addresses, as well as the pointers Copilot finds in the retrieved regions themselves, are all represented in in terms of the host kernel's virtual address space. Consequently, before making a DMA request on the PCI bus, the Copilot monitor must first translate these host virtual addresses into host physical addresses.

The Copilot monitor makes this translation by retrieving the page tables maintained by the host kernel's virtual

memory subsystem via DMA and using them to translate addresses just as the host kernel does. The nature of linear-mapped virtual addresses in the Linux kernel enables Copilot to overcome the chicken-and-egg problem of how to retrieve the host kernel's page tables when those same page tables are seemingly required to initiate DMA. This solution is described more fully below.

Figure 2 contains a diagram showing two of the three kinds of virtual addresses used by the host kernel and their relationship to physical addresses. On the 32-bit i386 platform, the Linux kernel reserves virtual addresses above `0xc0000000` for kernel text and data structures [2, 30]. Virtual addresses between `0xc0000000` and the point marked `high_memory` in the diagram are called linear-mapped addresses. There are as many linear-mapped addresses as there are physical addresses on the host; the point where `high_memory` lies may be different from host to host, depending on the amount of RAM each host has. These addresses are called linear-mapped addresses because the Linux kernel maps them to physical addresses in a linear fashion: the physical address can always be found by subtracting the constant `0xc0000000` from the corresponding linear-mapped address. This linear mapping is represented by the arrow A in the diagram.

All of the Linux kernel's page tables reside in this linear-mapped region of virtual memory. Consequently, the Copilot monitor can take the virtual address of the topmost node in the tree-like page table data structure from `System.map` and subtract `0xc0000000` to determine its physical address. It can then use this physical address to retrieve the topmost node via DMA. The pointers that form the links between nodes of the page table tree are also linear-mapped addresses, so the Copilot monitor can retrieve secondary nodes just as it did the first node.

This simple linear-mapped address translation method is sufficient to retrieve the host kernel text, its page tables, and those data structures statically-allocated in its initialized and uninitialized data segments ("data" and "idata" in the diagram). However, it is not sufficient for retrieving dynamically-allocated data structures such as the buffers containing LKM text.

These dynamically-allocated data structures reside in the region of host virtual memory running from the `high_memory` point to `0xfe000000`. The kernel does not map these virtual addresses to physical address in a linear fashion. Instead, it uses its page tables to maintain a non-linear mapping, represented by the arrow B in the diagram. In order to translate these virtual address to physical addresses suitable for DMA, the Copilot monitor evaluates the host kernel's page tables and performs the translation calculation in the same way the host kernel does.

The host Linux kernel in the Copilot prototype testbed organizes its memory in pages that are 4 kilobytes in size. Because of the linear nature of the address mapping

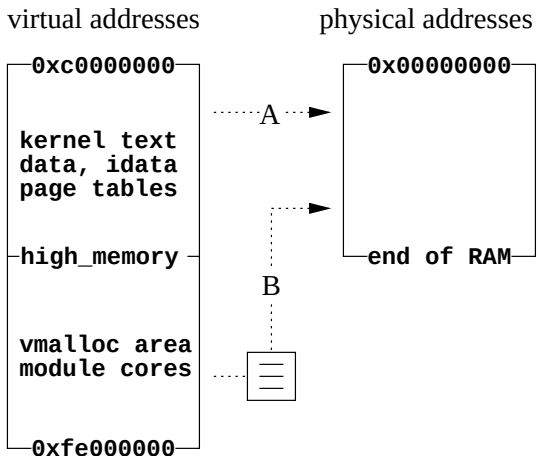

Figure 2: Virtual address translation

between `0xc0000000` and `high_memory`, the Copilot monitor is guaranteed to find large data structures that span multiple pages in this region of virtual memory stored in an equal number of contiguous page frames in physical memory. Because of this contiguous storage, the Copilot monitor can retrieve them with a single DMA transfer.

However, a single DMA transfer may not be sufficient for large data structures spanning multiple pages in the non-linear-mapped region of virtual memory. In this region, the host kernel's page tables may map pages that are contiguous in virtual memory onto page frames that are not contiguous in physical memory. Because of this potential for separation, the Copilot monitor must perform separate address translations and DMA transfers for each page when examining large data structures in this region.

Note that the Linux kernel's page tables cover only the part of the virtual address space reserved for the kernel; their contents do not depend on which processes are currently running. Note also that there are additional kinds of kernel virtual addresses not shown in the diagram. These are the persistent kernel mappings and fix-mapped addresses residing above `0xfe000000` in the kernel virtual address space. The Copilot monitor does not yet translate these addresses; they are not discussed here.

The Copilot monitor also relies on the absence of paging in kernel memory. Although the Linux kernel will sometimes remove pages of virtual memory belonging to a user process from physical RAM in order to relieve memory congestion, it never removes pages of kernel memory from physical RAM [2]. Consequently, regardless of which user processes are running or otherwise occupy physical RAM at the moment the Copilot monitor makes a DMA transfer, the kernel and its data structures will always be present in their entirety.

6 Integrity monitoring

This section describes the present Copilot prototype's strategy for monitoring the integrity of the host kernel. As outlined in Section 1's overview of the prototype, the Copilot monitor detects changes in critical regions of host kernel memory by hashing them. Because it looks for changes in general rather than symptoms specific to a particular set of well-known rootkits, the Copilot monitor can detect modifications made by new rootkits never before seen in the wild. The present Copilot monitor prototype hashes two classes of host kernel memory: (1) memory containing kernel or LKM text and (2) memory containing jump tables of kernel function pointers.

The reason for the first class is easily explained: by hashing all of the host's kernel and LKM text, the Copilot monitor can detect cases where a rootkit has modified some of the kernel's existing executable instructions.

The reason for the second class is more complex. Optimally, the Copilot monitor would be able to identify foreign text added into previously empty regions of host kernel memory by a rootkit, either via the kernel's standard LKM-loading interface or via its `/dev/mem` or `/dev/kmem` interfaces. Unfortunately, distinguishing between buffers of foreign text and buffers of harmless non-executable data is not easy on PC-compatible systems like the testbed host. The i386-family of CPUs do not provide a distinct "execute" permission bit for memory segments containing executable text; there is only a "read" bit used for both text and non-executable data [2].

Because of this difficulty, rather than attempting to identify the foreign text itself, the Copilot prototype monitors places where a rootkit might add a jump instruction or a function pointer that would cause the existing host kernel to execute the foreign text. According to the logic of this workaround solution, foreign text added to empty kernel memory is harmless provided that it cannot be executed.

The first part of the Copilot monitor's hashing strategy covers some of these cases by detecting places where a rootkit modifies existing host kernel or LKM text to jump to some foreign text. The second part describes what must be done to cover the rest: observe all of the host kernel's jump tables for additions and changes.

The Linux kernel has many such jump tables. Some, such as the system call table, are not meant to change during the kernel's runtime. Others, such as the virtual filesystem layer's file operation vectors, are designed to be amended and revised whenever an administrator loads or unloads a filesystem driver. Every LKM can potentially add more jump tables.

The only kernel jump table hashed by the present version of the Copilot monitor is the host kernel's system call vector – the most popular target of the rootkits surveyed in Section 2. Although this presently limited coverage provides many opportunities for clever rootkits to pass unnoticed, it is sufficient to demonstrate that host kernel integrity monitoring is possible from a coprocessor on a PCI add-in card.

7 Performance

In this section we present empirical results regarding the performance penalty of our Copilot system. As described below, Copilot has been shown experimentally to provide attackers with only a 30-second window while reducing system performance by less than 1%.

In recent years, it has become a well-known phenomenon that memory bandwidth creates a bottleneck for CPU efficiency [22]. While the addition of DMA to systems has increased performance by reducing the number of processor interrupts and context switches, it has also increased contention for main memory by allowing multiple devices to make requests concurrently with the processor. In the case of low-bandwidth devices, such as a simple network interface card (NIC), this additional memory contention has not proven to be significant. However, recent work has shown that the advent of high-performance network and multimedia equipment has begun to test the limits of the penalty for memory access [31, 32].

Because of the nature of our prototype, reading large amounts of memory periodically using the PCI bus, we fully anticipate some negative effect on system performance. We expect the degradation to be based on two primary factors. These are, in order of greatest to least significance, (1) contention for main memory and (2) contention for the PCI bus. The remainder of this section describes the penalties we have measured empirically through a set of benchmarks, the STREAM microbenchmark and the WebStone http server test suite. We conclude that while there is clearly a temporary penalty for each run of the monitor, the infrequency with which the system must be checked results in sufficient amortization of that penalty.

The STREAM benchmark was developed to measure sustainable memory bandwidth for high-performance computers [22]. While intended for these high-end machines and memory models, STREAM has proven to be an effective measure of PC memory bandwidth, at least for comparison purposes. The benchmark has four kernels of computation, each of which is a vector operation on a vector much larger than the biggest CPU-local cache. The four kernels can be summarized as a simple copy operation, scalar multiplication, vector addition, and a combination of the latter two.

To test the impact of our monitor on host memory bandwidth, we utilized a standard posttest-only control group design [4]. The experiment was run by bringing the system to a known, minimal state and running the STREAM benchmark. For the purposes of the STREAM tests, minimal is defined as only those processes required for system

Monitor Status	Average (MB/s)	Variance	Standard Error	Penalty
	COPY			
Off	921.997001	20.896711	0.144557	0.00%
On	833.016002	107.949328	0.328556	9.65%
	SCALE			
Off	920.444405	14.417142	0.120071	0.00%
On	829.142617	100.809974	0.317506	9.92%
	ADD			
Off	1084.524918	47.928264	0.218925	0.00%
On	1009.868195	86.353452	0.293860	6.88%
	TRIAD			
Off	1084.098722	49.922323	0.223433	0.00%
On	1009.453278	82.296079	0.286873	6.89%

Table 5: Summary of STREAM benchmark for 1000 runs with and without the monitor running.

operation, including a console shell. There were no network services running, nor cron, syslog, sysklog, or any other unnecessary service. We first ran STREAM 1000 times without the monitor running to obtain an average for each of the four kernels as control values. Similarly, we ran STREAM 1000 times with the monitor hashing in a constant while-loop. The monitor would therefore continuously read a configuration file for memory parameters, read system memory, make a hash of the fields it had read, compare that hash with a known value in a configuration file, report to the console the status of that hash, and continue with another memory region.

The results of our experiment, summarized in table 5, were verified using a standard t-test to be statistically significant ($p < .001$). Table 5 shows the computed mean, variance, and standard error for each group, separated by STREAM kernel. The fourth column, Penalty, is the percent difference of the average bandwidth with the monitor running and the average bandwidth without.

There are a few characteristics of the data worth noting. First, the greatest penalty experienced in this microbenchmark was just under 10%. We consider this a reasonable penalty given that the test environment had the board running in a continuous loop, a worst-case and unlikely scenario in a production environment. Second, it should be noted that the variance is significantly higher for the "monitor on" case for all four tests. This can be explained easily by the asynchronous nature of the two systems. Sometimes, for example when the monitor is working on a hash and compare computation, the host processor will immediately be given access to main memory. Other times, the host processor will stall, waiting for the board to complete its memory read.

The second benchmark utilized was the WebStone client-server benchmark for http servers. Similar to above, a standard "monitor on" or "monitor off" approach was taken to compare the impact of our prototype on system performance when the system is being used for a common task – in this case running as an Apache 1.3.29 dedicated web server. Additionally, we chose to test a third scenario, whereby the monitor was running, but at more realistic intervals. For the purposes of our experiment, we chose intervals of five, fifteen, and thirty seconds – numbers we believe to be at the lower (more frequent) extreme for a production system.

As with the STREAM benchmark, care was taken to bring the system to a minimal, consistent state before each trial. While the cron daemon remained off, syslog, sysklog, and Apache were running for the macrobenchmark tests. The experiment was conducted using a Pentium III laptop connected to the server via a Category 5e crossover cable. The laptop simulated 90-client continuous accesses using the standard WebStone fileset and was also brought to a minimal state with regards to system load before the test. The trial duration was 30 minutes and each trial was performed four times.

Table 6 clearly shows the "continuously running" tests each resulted in significantly less throughput (shown in Mb/s) for the Apache web server than the other four cases. Table 6 presents averages for the four trials of each monitor status (continuous, off, running at intervals). As can easily be seen from the data, running the monitor continuously results in a 13.53% performance penalty on average with respect to throughput. Fortunately, simply spacing monitor checks at intervals of at least 30 seconds reduces the penalty to less than 1%. As expected, the more frequently the monitor runs, the more impact there is on system performance.

We believe the data supports our original assessment that memory contention and PCI contention are the primary

Monitor Status	Average (MB/s)	Variance	Standard Error	Penalty
Off	88.842500	0.000158	0.006292	0.00%
30-second Intervals	88.097500	0.000892	0.014930	0.84%
15-Second Intervals	87.427500	0.000158	0.006292	1.59%
5-Second Intervals	85.467500	0.000158	0.006292	3.80%
Continuous	76.830000	0.002333	0.024152	13.52%

Table 6: Summary of WebStone Throughput results for 90 clients.

threats to performance. Because the Web server utilizes a PCI add-in card NIC, it is likely that the system was significantly impacted by PCI scheduling conflicts between the EBSA and NIC card, as well as memory contention between the EBSA/NIC (whichever got control of the PCI) and the host processor; note that the NIC driver utilizes DMA and so would also compete with the processor for memory cycles. We did not, however, perform any direct analysis of PCI contention and therefore cannot conclude this was the exact reason for decreased throughput.

The second, and more important, conclusion that arises from the WebStone data is that the penalty for running the system periodically is far less than that of running it continuously. Furthermore, since the monitor is meant to combat attackers who typically exploit vulnerabilities and then return days or weeks later to the system, the chosen interval of 30 seconds is an extremely conservative estimate for production systems. In addition, because of the configurability of the prototype solution, system administrators who are experiencing unacceptable performance losses can simply increase the interval.

8 Limitations

As described by Zhang [39], the fundamental limitation of a coprocessor-based kernel monitor is its inability to interpose the host's execution. For a PCI-based implementation such as Copilot, the view of the monitor is limited to main memory; there is no means of pausing the host CPU's execution or examining its registers. For this reason, Copilot will never be able to guarantee an invalid piece of code has not been executed. However, because Copilot can monitor main memory, the window of opportunity for an attacker to perform a successful attack without Copilot noticing is limited to timing attacks and extremely advanced relocation attacks. This section describes the difficulty of monitoring rapidly-changing host kernel data structures and both types of attacks that are currently feasible without Copilot detection.

Race conditions: Because the Copilot monitor accesses host memory only via the PCI bus, it cannot acquire host kernel locks as processes on the host can. Consequently, it may find host kernel data structures in an inconsistent state if its DMA requests are satisfied while a host process is modifying them. This limitation does not interfere with Copilot's ability to examine the static parts of the host kernel, such as its text and system call table. For data structures that change in response to relatively infrequent administrative actions, such as the host kernel's loaded LKM list, the Copilot monitor can compensate for the occasional inconsistent reading by reporting the trouble and repeating the examination.

However, for data structures that change much more rapidly during run-time, such as the host kernel's process table, repeated examinations seem unlikely to reveal the data structure in a consistent state. There would be some value in overcoming this limitation, as some rootkits modify the state of a host kernel's process table in order to conceal the presence of an attacker's processes.

Despite this limitation, the present Copilot monitor prototype manages to provide effective integrity monitoring functionality. Consequently, the cost of this potential for race conditions does not outweigh the value of the protective separation from the host provided by running the Copilot monitor on a PCI add-in card.

Timing attacks: The Copilot monitor is designed to run its checks periodically: every 30 seconds by default in the present prototype. A clever rootkit might conceivably modify and rapidly repair the host kernel between checks as a means of avoiding detection, although this lack of persistent changes would seem to decrease the utility of the rootkit to the attacker. In order to prevent such evasion tactics from working reliably, the Copilot monitor might randomize the intervals between its checks, making their occurrences difficult to predict [23].

Relocation/cache attacks: Copilot works because it has a picture of what an uncompromised system would look like under normal conditions. The fundamental assumption underlying detection of malicious modifications is that such attacks would result in a different view of the parts of memory monitored by Copilot. However, if an adversary were able to maintain a consistent view of Copilot's monitored memory while hiding malicious code elsewhere, such as in the processor cache, this code would remain unde-

tected by Copilot. However, it is currently unclear to what extent such attacks would succeed on a more permanent scale. Caches get flushed frequently and more extensive attempts to relocate large portions of the operating system or page tables would likely require difficult changes to all running processes. Future work will investigate the degree to which such highly sophisticated attacks are feasible without Copilot detection.

9 Future work

This section discusses a number of ways in which the present Copilot monitor prototype might be improved through future work.

Administrative automation: Version 2.4 and 2.6 Linux kernels provide many jump tables where LKMs can register new functionality for such things as virtual filesystems or mandatory access control. When Copilot monitors the integrity of such jump tables, it is designed to report any additions or changes it sees to the admin station, regardless of whether they were caused by a rootkit or by valid administrative action (such as loading a filesystem or security module).

As noted in Section 1, in the present Copilot monitor testbed, a human administrator is responsible for distinguishing between these two possible causes whenever a report arrives. For example, administrators might disregard a report of changes to the host kernel's security module operation vectors if they themselves have just loaded a new security module on the host. Future work might increase the level of automation on the admin station by implementing some kind of policy enforcement engine that will allow reports of certain modifications to pass (perhaps based on a list of acceptable LKMs and the data structures they modify), but act upon others as rootkit activity. Further improvements to the admin station might enable centralized and decentralized remote management of multiple networked Copilot monitors.

Filesystem monitor integration: The Copilot monitor prototype is the successor of an earlier filesystem integrity monitor prototype developed by Molina on the same EBSA-285 PCI add-in card hardware [23, 24]. Rather than examining a host's kernel memory over the PCI bus, this monitor requested blocks from the host's disks and examined the contents of their filesystems for evidence of rootkit modifications. Future work could integrate the Copilot monitor's functionality with that of the filesystem monitor, implementing both monitors on a single PCI add-in card. This integrated monitor could provide multiple layers of defense by monitoring the integrity of both the host operating system's kernel and the security-relevant parts of its filesystems.

Replacing corrupted text: As discussed in Section 3, some existing kernel-modifying rootkit detectors have the ability to replace certain parts of a corrupted system. These sections include certain jump tables, such as the system call vector, as well as kernel text. Based on the same DMA principles that give Copilot access to read system memory, it should be similarly possible to provide replacement of corrupted text through the same mechanism. We currently plan to investigate possibilities for kernel healing, particularly as it relates to the automated administration discussed above.

Known good hashes: Like many of the user-mode rootkit detection programs described in Section 3, the present Copilot monitor prototype generates its known good hashes by examining the contents of host kernel memory while the host kernel is assumed to be in a non-compromised state. This practice admits the possibility of mistaking an already-compromised kernel for a correct one at Copilot initialization time, causing Copilot to assume the compromised kernel is correct and never report the presence of the rootkit. This concern is particularly relevant in the case of LKMs, which Copilot hashes only after the host kernel loads them late in its runtime.

This limitation might be addressed by enabling the Copilot monitor to generate its known-good hashes from the host's trustworthy read-only installation media. This task is not as straightforward as it seems, however. The image of the host kernel's text stored on the installation media (and on the host's filesystem) may not precisely match the image that resides in host memory after host bootstrap. For example, the Linux 2.6 VM subsystem's Page Global Directory resides amid the kernel text. This data structure is initialized by the host kernel's bootstrap procedure and subsequently may not match the image on the installation media. Nonetheless, it may be possible to predict the full contents of the in-memory image of the host kernel from its installation media, perhaps by simulating the effects of the bootstrap procedure. Exploration of this possibility is a topic of future work.

Jump tables: As noted in Section 6, the Linux kernel has many jump tables. Any of these jump tables may become the target of a rootkit seeking to register a pointer to one of its own functions for execution by the kernel. Every LKM can potentially add more jump tables. The present Copilot prototype monitors the integrity of only a few. Future work can extend this coverage, perhaps in an attempt to cover the jump tables of some particular configuration of the Linux kernel, using only a specific set of LKMs.

10 Related work

This section provides a brief summary of work related to the Copilot monitor. As discussed in Section 9, the Copilot monitor owes a debt to earlier work by Molina [23, 24] that demonstrated the use of the EBSA-285 PCI add-in card as a filesystem integrity monitor. As with the Copilot monitor,

the use of the EBSA-285 allowed the filesystem monitor to operate correctly even in cases where the host kernel was compromised – an advantage over monitors such as Tripwire [18] that run on the host itself.

Copilot's integrity monitoring activities can be viewed as a kind of intrusion detection – its memory hashing technique may be likened to an attempt to distinguish between "self and non-self" in the host kernel [7]. Useful intrusion detection functionality has been demonstrated in a variety of ways [28]; while most of these techniques focus on detecting misbehavior in user programs rather than in the kernel, some gather information in kernel-space [33, 19].

Concurrently with Molina's work on coprocessor-based filesystem intrusion detection, Zhang et al. proposed using a secure coprocessor as an intrusion detection system for kernel memory [39]. Specifically, the authors describe a method for kernel protection that consists of identifying invariants within kernel data structures and then monitoring for deviations. This strategy of interpreting and comparing kernel data structures is very similar to that of Copilot.

However, there are a number of significant differences between Zhang's work and Copilot. Most notably, Zhang's design was not implemented on an actual coprocessor, but instead a proof-of-concept LKM was used to track critical kernel data structures. The use of an LKM allowed the authors to concentrate on determining which kernel invariants would be potential targets for a real implementation. While the authors propose using a PCI-based cryptographic coprocessor as the basis for their design, they fail to point out some of the inherent difficulties facing a real implementation such as virtual memory translation and bus-to-physical address translation. In addition, without having implemented the design, the authors conclude that the number of detection samples will be limited by the processing power of the coprocessor. As discussed in Section 7, experiments have suggested that contention for the bus and main memory are more likely to be the limiting factor.

Many other projects have explored the use of coprocessors for security. The same separation from the host which allows the Copilot monitor to operate despite host kernel compromise is also useful for the protection and safe manipulation of cryptographic secrets [38, 12, 14, 20].

There have been at least two demonstrations of probabilistic techniques for kernel integrity-monitoring that operate without the use of additional hardware such as Copilot-style PCI add-in cards. These techniques involve a monitor host that sends computational challenges to a target host. The target host must run a verification procedure to calculate the correct response within a carefully-calculated time limit. The verification procedure is structured in a manner that makes it probable that a compromised target host will either return an incorrect result, or take too long in coming up with a convincing lie. Determining the proper time limit requires intimate knowledge of all aspects of the target host's hardware that might effect the speed of its verification procedure calculation, from the timing of various instructions to the size of the CPU's cache and translation lookaside buffer.

One of these techniques, SWATT [34], is designed to monitor embedded devices without virtual memory support and is consequently not applicable to the virtual memory-using PC-architecture GNU/Linux hosts targeted by Copilot. Furthermore, because of the nature of the SWATT verification procedure, if it were run on a PC-architecture host, it would have to hash the host's entire physical memory, rather than just those parts which contain static text and data. Consequently all of the host's dynamic data, including runtime stacks and heaps, would need to be in a known state in order for the calculation to return a correct result. Although this requirement may be reasonable on some embedded devices, it would be unrealistic for the hosts targeted by Copilot.

Kennell and Jamieson [17] have demonstrated a related technique that, unlike SWATT, is designed to monitor the same PC-architecture GNU/Linux hosts as Copilot. Their technique requires the target host to run a specially-modified Linux kernel that includes a verification procedure at the head of its linear-mapped static text segment. This verification procedure is capable of probabilistically verifying its own integrity and the integrity of the static kernel text. The authors predict that this initial verification procedure could demonstrate the integrity of an additional second-stage verification procedure also implemented in the static portion of the kernel text. This second-stage verification procedure might use a different algorithm to verify the integrity of the dynamic portions of the kernel that are beyond the initial verification procedure's reach.

When compared with Copilot, Kennell and Jamieson's technique possesses the advantage that it requires no additional hardware. Copilot, on the other hand, is capable of monitoring unmodified commodity Linux kernels. Both techniques appear to provide effective integrity monitoring. However, the Kennell and Jamieson verification procedure must disable hardware interrupts during its computation. Their prototype benchmark results show an entire challenge, compute, encrypt, response dialog taking 7.93 seconds on a 133MHz Intel Pentium-based PC. Because the effectiveness of the technique depends upon the compute portion dominating the time taken for the entire dialog, this suggests the PC spent the bulk of this time with interrupts turned off. Kennell and Jamieson do not provide benchmark results showing the effect of their technique on overall application performance; a comparison with Copilot's results in section 7 is consequently not possible.

Investigations into secure bootstrap have demonstrated the use of chained integrity checks for verifying the validity of the host kernel [15, 1]. 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. The Copilot monitor operates after host kernel bootstrap is complete and provides evidence that the host kernel remains in a desirable state during its runtime.

There are many software packages intended to detect the presence of kernel-modifying rootkits, including St. Michael and St. Jude [16]. However, these software packages are intended to run on the host that they are monitoring and will operate correctly only in cases where a rootkit has not modified the behavior of the kernel's `/dev/mem` and `/dev/kmem` interfaces to hide its own presence. Because it runs on a coprocessor on a separate PCI add-in card, the Copilot monitor does not share this dangerous dependence on the correctness of the host kernel and can be expected to operate correctly even in cases where a rootkit has arbitrarily modified the host kernel.

Kernel-resident mandatory access control techniques represent a pro-active alternative (or compliment) to the reactive approach of the Copilot monitor and other intrusion detection mechanisms. Rather than detecting the presence of a rootkit after an attacker has installed it on a host, these techniques seek to prevent its installation in the first place [36]. There have been many demonstrations of mandatory access control on the Linux kernel [6, 8, 25, 21, 27, 10]; the latest versions of the Linux kernel contains a Linux Security Modules interface specifically to support these techniques [37]. However, even systems with extensive mandatory access controls can fall prey to human failures such as stolen administrative passwords. Copilot is designed to detect rootkits in hosts compromised by such failures.

11 Conclusion

The Copilot project demonstrates the advantages of implementing a kernel integrity monitor on a separate PCI add-in card over traditional rootkit detection programs that run on the potentially infected host. Because the Copilot monitor software runs entirely on its own PCI add-in card, it does not rely on the correctness of the host that it is monitoring and is resistant to tampering from the host. Consequently, the Copilot monitor can be expected to correctly detect malicious kernel modifications even on hosts with kernels too thoroughly compromised to allow the correct execution of traditional integrity monitoring software. The Copilot monitor does not require any modifications to the host's software and can therefore be easily applied to commodity systems.

The Copilot monitor prototype has proven to be an effective kernel integrity monitor in tests against 12 common kernel-modifying rootkits. In its default configuration, the Copilot monitor prototype can detect changes to a host kernel's text, LKM text, or system call vector within 30 seconds of being made by a rootkit. Its monitoring activities do not require the consent or support of the host kernel and cause minimal overhead: for example, less than a 1% throughput performance penalty on a 90-client WebStone webserver benchmark. Because its hashing-based approach detects changes in general, rather than focusing only on specific symptoms of a well-known set of rootkits, the Copilot monitor can detect both known rootkits and new rootkits not seen previously in the wild.

References

[1] W. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., second edition, December 2002.

[3] D. Brumley. invisible intruders: rootkits in practice. *;login: The Magazine of USENIX and SAGE*, September 1999.

[4] D. T. Campbell and J. C. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Company, 1963.

[5] A. Chuvakin. Ups and Downs of UNIX/Linux Host-Based Security Solutions . *;login: The Magazine of USENIX and SAGE*, 28(2), April 2003.

[6] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference*, New Orleans, Louisiana, December 2000.

[7] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, Oakland, California, 1994.

[8] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Berkeley, California, May 2000.

[9] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Berkeley, California, May 1999.

[10] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

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

[13] Intel Coporation. 21285 Core Logic for SA-110 Microprocessor Datasheet, September 1998. Order Number: 278115-001.

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

[15] N. Itoi et. al. Personal Secure Booting. *Lecture Notes in Computer Science*, v. 2119, 2001.

[16] K. J. Jones. loadable kernel modules. *;login: The Magazine of USENIX and SAGE*, 26(7), November 2001.

[17] R. Kennell and L. H. Jamieson. Estabilishing the Genuity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–310, Washington, D.C., August 2003.

[18] G. H. Kim and E. 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*, pages 18–29, Fairfax, Virgina, November 1994.

[19] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecing and Countering System Intrusions using Software Wrappers. In *Proceedings of the 9th USENIX Security Symposium*, pages 145–156, Denver, Colorado, August 2000.

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

[21] P. A. Loscocco, S. D. Smalley, and T. Fraser. 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.

[22] J. D. McCalpin. Sustainable Memory Bandwidth in Current High-Performance Computers, October 1995.

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

[24] J. Molina and W. 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.

[25] A. Ott. Rule Set Based Access Control (RSBAC) Framework for Linux. In *Proceedings of the 8th International Linux Kongress*, Enschede, November 2001.

[26] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann Publishers, second edition, 1998.

[27] N. Provos. Improving host security with system call policies. Technical Report 02-3, CITI, November 2002.

[28] L. R., D. Fried, J. Haines, J. Corba, and K. Das. Evaluating intrusion detection systems: Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In *Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection*, October 2000.

[29] R. Rivest. The MD5 Message-Digest Algorithm. Technical Report Request For Comments 1321, Network Working Group, April 1992.

[30] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., second edition, June 2001.

[31] S. Schonberg. Impact of PCI-bus load on applications in a PC architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 430–439, Cancun, Mexico, December 2003.

[32] S. e. a. Schonberg. Performance and Bus Transfer Influences. First Workshop on PC-based System Performance and Analysis, October 1998. San Jose, California.

[33] R. Sekar and P. Uppuluri. Synthesizing Fast Prevention/Detection Systems from High-Level Specifi cations. In *Proceedings of the 7th USENIX Security Symposium*, Washington, D. C., August 1999.

[34] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.

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

[36] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confi ning Root Programs with Domain and Type Enforcement. In *Proceedings of the 6th Usenix Security Symposium*, pages 21–36, San Jose, California, July 1996.

[37] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th Annual USENIX Security Symposium*, pages 17–31, San Francisco, California, August 2002.

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

[39] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.