Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System

Lifu Wang
Department of Computer Science and Engineering
Arizona State University
Tempe AZ, 85281

Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ, 85281

Abstract

Malware and rootkits are serious security vulnerabilities, and they can be designed to be resistant to anti-viral software, or even totally undetectable. This paper described a hierarchical trust management scheme, where the root of trust is in a non-tamperable hardware co-processor on a PCI bus. The hardware checks a part of the OS kernel for integrity which in turn checks other parts till we ensure the entire system is free of rootkits. The checker can be extended to encompass all the applications and anti-virus software. Our system can detect illegal modifications to kernel, loadable kernel modules and user applications. It also provides a secure communication line for user interaction to enable legal software updates. Our tests show that we can correctly detect different real-world and synthetic rootkits even though the host kernel is compromised.

1 Introduction

Malware attacks have become a serious threat over past few years. They spread over the Internet quickly, as computers get prolific online. A survey by Cybertrust's ICSA Labs [1] found that from January 2004 through December 2004, a rate of virus infection increased 116% month-by-month. In Jun 2006, F-Secure Inc. reported that over 185,000 viruses exist [2]. The frequency and cost of malware attacks have increased for ten consecutive years. McAfee claimed [3] malicious threats are emerging each month, increased from 300 per month two years ago to 2,000 per month today. McAfee also showed [3] that malicious threats are evolving into hybrid versions that contains viruses, rootkits, worms, and DoS attacks together. Malware authors usually do not write malicious code for fun. Estimated 85% of malware are written purely for profit. The malware steal computer owner's digital identity, credit card information or bank account numbers. Also they use the victim's machine to send SPAM emails. Security experts warned that consumers are getting fearful of using computers due to the security concerns.

Amongst malware, rootkits are the most dangerous threat. They are particularly difficult to detect and prevent, because they are internal to the operating systems and hide by patching the kernel. For example, rootkits can simply disable anti-virus tools or hide the processes, files and routines that compose the viruses. By exploiting software vulnerability such as buffer overflow, intruders can gain control over the host computer and install rootkits. After a vulnerability is discovered, rootkits writers can have attacks ready in a few days today, as opposed to 11 months in 2001 [4]. The rootkits technology has been mature enough in past years and now not only hackers but also commercial software companies are using it. Recent Sony BMG's XCP software was found it employed rootkits [5] to hide their digital right management files and processes so the user cannot disable it or make illegal copies of their music. Within months, it was discovered even anti-virus company had used rootkits to hide a directory for their virus detection purposes. In the past three years, the use of rootkits technologies have grown by more than 600% [6]. Same study also found that even though rootkits have become more sophisticated, it is now easier to write and spread rootkits.

Unfortunately, there is no real solution to malware detection. It is a hard problem. Fred Cohen [7] has proven that mathematically perfect detections of unknown viruses is equivalent to the halting problem. It means that no arbitrary algorithm can look at other programs and determine either "a malicious logic is present" or "no malicious logic is found" in finite time, also known as *Turing Undecidability*. Current anti-virus or anti-malware software are just a workaround. They cannot detect new viruses and they cannot detect new designed rootkits at all.

In this paper, we present an integrated hardware-software co-design solution. It is designed to monitor the integrity of running software in memory from outside the machine. It also allows legal software updates. At the lowest level is a

new PKI-based (Public Key Infrastructure) hardware device added into host computer. This device contains two components - the "security core" called *SecCore* and "secure I/O" called *SecIO*.

The SecCore consists of hardware connected to the host computer via PCI bus and software running on this hardware. The SecCore hardware can access the main memory of the host computer and can also interrupt the host CPU. This allows the software running on SecCore to act like the root-of-trust aside the host operating system. It checks a critical routine in the kernel and then relies that routine to build up a hierarchical trusted software chain. The SecIO ensures that the SecCore can communicate with a user without any malware disrupting or spoofing the SecCore outputs message or user input. We will describe the SecIO and SecCore in more detail in section 5.

The SecCore bears some similarity to TPM and Copilot. TPM (Trusted Platform Module) [8] chip along with the LT (LaGrande Technology) [9] motherboard and NGSCB (Next Generation Secure Computing Base) [10] secure operating system is a security solution from TCG (Trusted Computer Group) [11]. The TPM may be capable of secure bootstrap but subsequent deployment of malware can go undetected. The TPM also has no secure IO, which is a severe shortcoming. Copilot is a hardware coprocessor that constantly monitors the host kernel integrity [12]. It cannot handle the dynamic kernel modules and user-level applications. It does not have a mechanism addressing the kernel update issues. Our hierarchical checkers can not only attest the running kernel but also modules and applications. In addition, our SecIO is a special design for legal software update.

The goal of this paper is to raise the severity of rootkits threats and propose a solution. The remainder of this paper is structured as follows. Section 2 to 4 describe and define our motivation, threat model and related work. Section 5 discusses the proposed solution. Section 6 to 7 present our implementation and simulation. Section 8 discusses the limitations and open issues. Finally, section 9 concludes our work.

2 Motivation and Challenges

In this paper we offer a solution to detect illegal software modification at run time. This scheme must constantly monitor all of running software in memory, including operating system, kernel modules and applications to assure their integrity. It also must be able to be flexible enough to allow legal update or re-installation. When using the computers, users should be given confidence that their computers run the software that is unchanged from initial sources.

Anti-virus software can detect some existing viruses, they suffer from many limitations. For example, they cannot detect new viruses, they cannot detect kernel rootkits and themselves can be tampered with, disabled and the virus database can be forged. In addition, they have other side effects such as performance slowdown due to the growing size of innumerous virus database. Finally, the false positive alarms in virus detection are usually annoying, but false negative alarms could be very dangerous.

There are other cryptographic approaches using shared secrets and PKI algorithm to protect software and data. A shared secret scheme is easily compromised because the secret is managed by software and OS. A PKI scheme by itself does not provide a complete solution. If it is not properly implemented, the private key can be stolen too. Current shared secret and PKI sometimes suffer similar vulnerabilities as anti-virus software. Even if they are implemented in kernel, rootkits can attack the binary by rewriting the raw memory.

3 Threat Model and Goals

Any security solution must live up to a threat model. In this section, we discuss the conventional Internet Threat Model, the Thompson Threat Mode, and our Viral Threat Model.

In *Internet Threat Model*, an attacker can get complete control of the communication media between two ends (client and server), but the hosts running applications and protocols are not compromisable. It assumes adversaries living on the Internet can sniff, forge, spoof or relay the data during transmission. Current defense schemes including IPsec (IP Security), TLS (Transport Layer Security), SSL (Secure Sockets Layer) and S-HTTP (Secure HTTP) prevent the threats in the Internet Threat Model.

On host computers, every activity is just software-driven. Strictly speaking, we cannot trust any software and bootstrap that run on hardware by the user, shown by Ken Thompson [13]. He suggested no amount of source level verification or scrutiny will protect us from malicious code. We call this the *Thompson Threat Model*. Since it is impractical for anyone to create entire software stack, assembler, compiler, kernel and libraries from scratch, this threat model is too strict to have any feasible.

We define our threat model as *Viral Threat Model*. It is a looser form of the Thompson Threat Model. In order to enable the use of commodity software on host computers we assume the presence of some trusted software. In our case, these are the software in our security hardware and those initial software from vendor. In Viral Threat Model, malware penetrates or sneaks into host computer such that it can gain control over the system as well as infect other software.

Like other solutions to criminology in our society, there is no single silver bullet that can totally stop all kinds of

crimes. But if there exists a solution that can stop most of the threats in a relatively efficient manner, the solution is practical. Our goal is to provide a mechanism to attest the integrities of running software in the Viral Threat Model. Our goals do not cover network attacks such as DDoS (Distributed Denial of Service), IP spoofing, MAC masquerading, or network penetration. We do not address on digital right management and resource stealing issues either.

4 Related Work

Many of early OS security work addressed building security kernels [14] [15] [16], but there never existed a real system. Making security kernels are infeasible as the size and complexity grow in modern commodity operating systems because of their rich features. In recent years, security hardware is becoming mainstream. It is eventually realized software only security may be not adequate to stop malware [17]. Analysts from IDC have forecasted by 2007, 80% of computers will be equipped with hardware-based security devices rather than security software.

Hardware-based security [18] has been around since mid-1990 but it has been slow to catch up due to extra expensive hardware cost and lack of application support. There are other hardware security devices such as IBM's PCI crypto card [19], Intel's Processor ID [20], Intel's encrypted CPU instruction sets [21], UltraSPARC's serial numbers [22] and so on. None of above work is widely accepted by consumers. One of the reasons is the lack of third party software support.

A secure and reliable bootstrap architecture [23] uses the AGES modified security platform to monitor the bootstrap with a trust chains. The scheme is successful, but it only limits to bootstrap and does not handle running system. TCG (Trusted Computer Group) [11] [24] finally merged major vendors and their work into an alliance and TPM (Trusted Platform Module) [8] is its security hardware specification. Following the specification, Intel introduced LT (LaGrande Technology) [9] platform with TPM chip integrated into the chipset. NGSCB (Next Generation Secure Computing Base) [10] [25] from Microsoft is a software architecture that runs on the TPM and LT. The future of TPM is not clear due to its complexity, and critics who points its primary usage may be to enforce DRM (Digital Right Management) [26] [27]. Our work is different from above technology in different ways. First, we do not re-design a new VMM-like trusted OS like NGSCB. Second, we do not separate applications into trusted and untrusted land. Third, our design is much simpler.

The Copilot [12] is another hardware coprocessor-based solution that supports a kernel integrity monitor for commodity systems. It can detect malicious modification even a host kernel is thoroughly compromised. The Copilot is a more hardware-oriented approach than ours. Thus it has no knowledge about dynamic loadable kernel modules and applications at run time. Beside, the Copilot cannot distinguish an update is a user-driven or rootkit-driven, so it cannot securely perform an legal update. Our work is a more hardware-software balanced approach, that can check the running modules and applications as well as allow legal updates to be performed securely.

5 Our Approach

In this section, we present a high-level overview of our solution. We first introduce the internal details of the SecCore and SecIO in section 5.1. The SecCore contains hardware to access host memory, store PKI keys and run software for computing crypto functions such as MD5 and RSA. It can access the system memory and interrupt the host CPU. The SecIO is physically attached to SecCore and it is not accessible by any host software. Section 5.2 discusses the mechanism that uses the SecCore to check the integrity of the running host kernel. The key point is similar to Copilot that SecCore is independent and cannot be disabled or controlled by the host CPU. Then we present some deficiency of a kernel checker implemented externally like Copilot. We address this problem in section 5.3 using hierarchical checking. We install a new interrupt handler in the host kernel that can be directly checked and called by the SecCore. We also install a kernel and application checker inside the host kernel. These checkers will be verified and called by that interrupt handler. In other words, the hierarchy starts from the root-of-trust SecCore hardware, expands up towards the interrupt handler and software checkers, and it continues until all other software are assured and trusted. In addition, our system does not only attest software on the host computer, but it can adapt to legal software updates by using the SecIO facility, which will be presented in section 5.4. In section 5.5, we show a number of viral attacks and threat analyses.

5.1 SecCore and SecIO Component

The SecCore is an embedded system that runs an operating system and software, and is invoked by a timer. As a hardware security device, the SecCore must be secure enough to be unconditionally trusted. This claim is based on the fact that the functions are residing and executing in the separate tamper-resistant hardware inaccessible by host software. Figure 1 shows the SecCore is a PCI device that can be plugged into PCI expansion slot. The SecCore shares common the basic features that are found in other crypto chips such as TPM. Unlike other crypto chips, the SecCore requires the following three conditions.

- Its internal resources are not accessible by the host chipset or CPU.
- It must be able to access a part of host system memory.
- It can halt or suspend the system whenever necessary.

The SecCore consists of a microprocessor, memory, timer, PCI-to-PCI bridge and I/O controller. A low-end microprocessor can execute instructions. There are different kinds of memory blocks. Programs and key-pairs are stored in non-volatile read-only memory, and the running code and data are held in random-access memory during computation. It has a timer to generate a heartbeat. Each heartbeat activates the SecCore and it checks the host software and signals the host CPU. An I/O controller allows the SecIO physically connected to the SecCore's I/O port. A PCI-to-PCI bridge provides an interface between the SecCore to the host chipset.

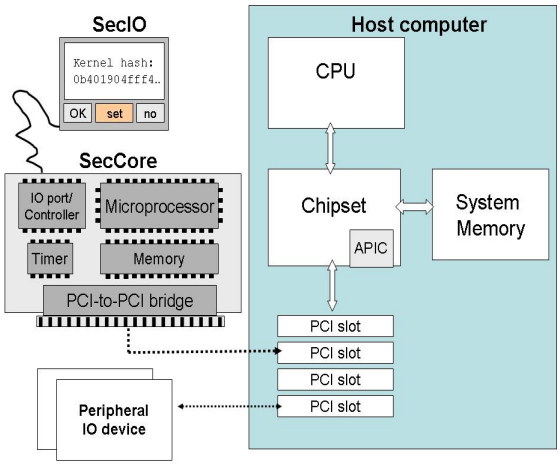

Figure 1. The Security Component in Platform

The SecCore runs a small operating system, only with basic kernel components, and applications. These functions include the PCI protocols, I/O drivers and crypto functions. They can be fabricated by hardware, firmware or hybrid. Direct hard-wired circuits have the best performance while a firmware approach trades off performance for flexibility, complexity and updatability. A hybrid is the middle ground approach like programmable hardware. FPGA (Field Programmable Gate Array) is a popular technology. The design adds semiconductors programmed to duplicate basic logic gates, math or combinatorial functions. It is widely used in DSP (Digital Signal Processing) processor and network TOE (TCP/IP Offload Engine) chip. A large amount of crypto functions are well-defined mathematic computations, thus FPGA is a good implementation choice. All functionalities must be pre-programmed by the manufacturer, and even though technically speaking, modifications or updates by users are possible, but this ability should be disabled because this is a security risk.

From the host operating system side, when the SecCore is plugged in and discovered, the operating system allocates a non-maskable interrupt vector, configures its software-mapped address, requests an I/O region, enables DMA line and creates a device special file. These are the standard configurations for all PCI peripheral devices. Communications between the host and SecCore are bi-directional. For host-to-SecCore communication, the host requests a service by writing to SecCore's command register and data are passed via its PCI memory. The SecCore will update its status register when an operation is completed so the host can read. These operations typically done by `open`ing the SecCore device file followed by the `ioctl` system call. For SecCore-to-host communication, the SecCore asserts its interrupt pin to deliver a signal to the host CPU when it needs attention. The data transfer is the same as in host-to-SecCore request.

In the figure 1, we show the SecIO is connected to the SecCore. It is a small input/output device that can actually be any kind of inexpensive I/O hardware ranging from a tiny mono-color display and calculator-style keypad to higher end TFT touch-screen. These devices are very common. Its keypad contains a set of few special buttons - an "ok", "no" and "set". We assume all I/O to and from SecIO is "secure" that is not visible and tamperable by the host software.

5.2 Checking Kernel Integrity

A checker verification method is the computation of a hash upon initialization, which is then regularly re-checked to ensure there has been no modification of the software. By integrity, we mean the integrity of host kernel's code area, known as `.text` segment. We however do not check the entire `.text` region from SecCore and the reason for this modification is discussed later in this secton. Software integrity is a distinct identity presented by different ways such as checksum, SHA or MD5 algorithm. Using integrity technique to protect data is quite mature and prevalent already. However, most current identity checking schemes are dealing with files. The purpose is to prevent data from being corrupted during transmission or on storage since the checksum was computed. These techniques cannot efficiently defend against the viral threat model. In this paper, we assume software vendors may optionally provide a certificate and integrity of the `.text` area for their software and this initial integrity can be directly used by checkers. If vendors do not provide anything, the owner can still compute an initial integrity on the host computer.

The SecCore must periodically execute its checking

function against kernel `.text` to ensure it is not been altered or destroyed in an unauthorized manner. Checking integrity of a running kernel is more complicated than checking integrity of a file as discussed later. We first briefly discuss the Linux ELF (Executable and Linkable Format) and PCI memory addressing, because they are primary key mechanisms used in kernel checking. Overall, every software, including a kernel and applications, are ELF files. It is made up of one program header followed by a number of section headers, such as `.text`, `.data`, `.bss` and `.symtab`. These headers hold all information such as its name, type, size, file offset, memory image starting address and so on. In our case, we are mainly concerned about the `.text` section because it is loadable and it contains the static instructions. By default, kernel `.text` starts from virtual address `0xc0100000` on x86 platform. It means bootstrap loads the kernel image at virtual address `0xc0100000`, that is the physical address `0x00100000`. With the information about the address, its size and initial integrity, the host CPU can locate and check the kernel image in memory. Later we show why this is not an effective way for kernel integrity checkers.

As stated earlier, the SecCore is able to access the host memory. That is done by mapping PCI-shared host memory into SecCore's space. From the SecCore's point of view, the host memory is like another peripheral device's memory buffer that can be directly mapped into its own I/O space. Thus, using the PCI standard, the SecCore can assign a base address to the host address decoder. This is the way to enable the SecCore can access the kernel `.text`.

Using SecCore to monitor the running kernel has two limitations. It cannot verify modules and it cannot verify user processes. First, since the SecCore must know the predetermined physical address and its size of `.text` segment, that means it can only handle static kernel `.text`. Modern operating systems are designed to keep the base kernel as small as possible while other services are put in modules. Modules, or LKM (Loadable Kernel Modules), are object files that contain some code to extend the running kernel. A module is not an executable, it cannot be run standalone, and it does not have an initial integrity. It is not possible to verify a module because the SecCore does not know where the module was loaded by kernel and what its integrity was. Second, most system services are implemented as root-privileged applications and they automatically start after system is up. The SecCore does not know the host kernel process table and page-mapping table, so it cannot verify any user applications. It is impractical to make the powerless SecCore run complicated software or maintain the host kernels data structure dynamically.

5.3 Hierarchical Checking Against Software Integrity

We extend the checking mechanism into a hierarchy such that every running software can be covered. The concept is straightforward and shown in Figure 2. The SecCore is "root-of-trust hardware" device at the bottom. Instead of attesting the entire host kernel `.text`, the SecCore only verifies a small but critical block in kernel `.text`. This part, which we call *SecISR*, is an interrupt service routine that will be executed by host CPU as it receives a signal from SecCore. At this point, the SecISR becomes "root-of-trust software". The SecISR is actually a starter routine - it validates and then executes a *kernel checker* and *application checker*, which are another kernel functions. The *kernel checker* and *application checker* then become the next trusted software in the hierarchy. The kernel checker monitors the integrity of the entire kernel `.text` and modules, while the application checker verifies the integrity of some running processes, such as anti-virus software. Building up a trusted hierarchy always requires two operations - validation, if passed, followed by execution. In the figure, a solid arrow represents an execution flow and a dotted arrow represents a validation flow.

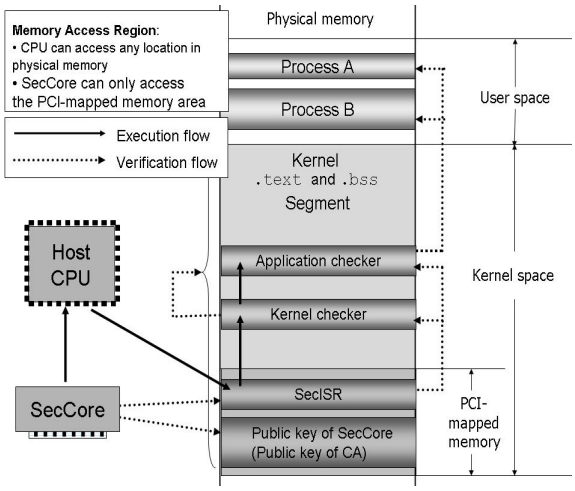

Figure 2. Hierarchical Checking

There are several advantages in our hierarchical checking approach. First, the SecCore only knows the SecISR and nothing else. Since the SecISR is inside kernel `.text`, its address, size and integrity can be pre-determined and kept in SecCore. Second, all checkers are software and they are performed by host CPU. But the hierarchical checking ensures that the checkers in the kernel cannot be modified or disabled. In addition, it is better to offload the work from SecCore to host CPU since a SecCore is much less powerful. Third, even though the SecCore shares host system

memory, it does not necessary mean the SecCore is allowed to address any location out there. For example, a common allowance in PCI-mapped memory is usually no more than 64KB. It means a host will allow 64KB of lower physical memory sharable. The size is too small for SecCore to look up all running software. In hierarchy checking, the only requirement is the SecISR must be in the PCI-mapped memory region, while other checkers and applications can reside anywhere in memory.

Above scheme has one drawback. Because the checkers run on host, the integrity data must be accessible by host CPU, rather than SecCore. These data, both in file and in memory, are vulnerable. As long as the checkers can access them, so can rootkits. To protect the data from being tampered, the SecCore must digitally sign them using its private key. Then the corresponding public key is exported to host so that the signature can be validated at later time. However, it leads another vulnerability that the public key can be replaced by a fake one and signature can be forged. To solve the problem, we need to better embed the public key inside the kernel and let the SecCore directly verify it. In figure 2, it shows the public key of the SecCore and CA are stored in kernel's `.bss` segment in lower memory. The kernel declares it an un-initialized global variable and during boot, the kernel reads the public keys from a file into this variable. Applying the same technique in checking SecISR, we can have the SecCore locate the address of this variable and compare with its own copy. Because all checkers are programmed to use this kernel variable for public key which is protected by the SecCore, therefore the checkers can securely validate the signature accordingly.

Each SecCore device comes a unique PKI key-pair and the public keys of CAs. The initial integrity setup should run at the host operating system post-installation time. As stated earlier, the vendor may have already provided the integrity data. In this option, the CA's public key is used by host software to validate the certificate. Otherwise, the owner can compute the initial integrity of the SecISR and kernel `.text` locally. These integrity data are passed to the SecCore by writing a request to its command register. The SecCore displays the information on its SecIO's display and it waits for the owner to confirm. For the integrity of the SecISR `.text`, the data is stored in SecCore, and for the integrity of the kernel `.text`, the data is signed and returned to host via its status register and PCI memory. Finally, the setup process requests the SecCore to export its public keys so it can save them into a file. After installed, the SecCore can operate on its heartbeats.

5.4 Software update

Software updates are very common today. Every security scheme must be flexible enough to adapt any legal update, but unfortunately, there is no easy solution. From the SecCore's point of view, it returns a signature of some integrity data when a host application requests so. For example, an owner patched anti-virus software and the installer computed the new integrity and requested a signature. At next moment, a rootkit might infect the software, re-compute its integrity and issue another request to the SecCore. If the second case is accepted, the viral software is totally legitimate and the application checker will never report anything. The problem is, how the SecCore can possibly distinguish between a rootkit-initiated and user-initiated request. A human-in-loop action is the most straightforward way to resolve it - unless the owner say a yes, the SecCore will never accept anything. However, if an user interaction is handled by host software, it brings another vulnerability - the output and input message may be spoofed by a compromised I/O driver. Even though the owner entered a no, the message will be always forged into a yes. Thus, the SecCore has no way of knowing the message is from the real owner and I/O driver or not.

The SecIO is brought to guarantee a genuine I/O message. Since the SecIO is directly connected to SecCore, it provides an out-of-band secure communication channel. Whenever the SecCore receives a request, it always shows the data on its display. The host application can verify the certificate from vendor, if available, but because the SecCore does not have a big display, the validation is actually done on host. The information will better assist the owner.

Using SecIO and human-in-loop action seem to cause extra workload, since the owner must exceptionally interact with the SecIO. In fact, the impact should not be drastic for the following reason. Our goal of the hierarchical checking is not to verify every arbitrary process like `a.out`. Instead, at application level we intend to assure those critical software and system services such as anti-virus or `xinetd`. That means only those anti-malware and system applications plus the kernel, checkers, and SecISR are in the hierarchy list. Other software can be scanned by anti-virus or anti-malware tools. Our approach does not replace the current anti-malware techniques at all. It co-exists and cooperates with these tools.

5.5 Attacks and Analysis

We now discuss different types of viral attacks. The first type is an attack or attacks to application binary. If a virus infects an application, both the virus and infected application will be detected by anti-virus tool. If a virus rewrites the binary of an anti-virus tool, once the tool is loaded into memory, it will be detected by the application checker. If a rootkit replaces any kernel routine in order to hide the presence of viruses, this activity will be discovered by kernel checker. If a rootkit disables the checking by removing the

SecISR, it will be immediately identified by the SecCore. Even if an attacker launches multiple attacks at same time and tries to break the chain, it cannot be successful as long as the SecCore is functional.

The second type of attacks compromises the initial integrity data. A rootkit cannot alter the public key of the SecCore, because it is in kernel `.bss` area and directly checked by the SecCore. A rootkit cannot tamper with the integrity data since they are protected by SecCore's digital signature. It is impossible to forge a signature and replace the public key in memory without be detected by SecCore. It is also impossible to trick the software checker reading another fake public key elsewhere without modifying its binary.

The third type of attacks makes the system accept some illegal updates. A rootkit cannot trick the SecCore sign anything unexpected, since the owner's confirmation is always requires. A rootkit cannot forge an output message to host display and trick the owner to sign a different integrity, because the message shown on host display and SecIO are different. Finally, a rootkit cannot forge an owner's input, because it has no way of controlling the SecIO.

6 System Implementation

Our host computer is a widely used PC with Pentium 4 2.0GHz CPU, 512 MB SDRAM, Intel 845G chipset and PRO/100 NIC. The security subsystem is simulated by *SlotServer 3000*, from OmniCluster Inc., due to its availability. The product was originally designed to allow users to add functionality to existing servers by placing a fully functional computer inside the host, using the PCI bus channel to the host. This particular model is based on Intel Pentium III CPU and VIA Apollo Pro266T chipset. The north bridge chip connects 2xAGP video (16MB video RAM), and 256 MB system memory, while the south bridge chip supports USB ports, IDE controller, super I/O chip, 10/100 Ethernet chip and audio. The SlotServer is a single board computer and mainly used for web server, firewall or distributed systems.

The configuration is close enough to our proposal platform. In a single box, it has two independent running systems, two sets of I/O devices (one physically attached to SlotServer), and shared resources through PCI bus. The host chipset could map the lower 64KB of SlotServer's main memory, but on reverse side, the SlotServer could only map 4KB of host's memory due to its chipset. We do not have any tools to reset the shared PCI memory allowance in chipset, however, because it is a two independent symmetric systems, our workaround is to use SlotServer as the host computer and use the Pentium 4 machine as the SecCore and SecIO. The SecCore runs Linux 2.4.32 kernel and the host operating system runs Linux 2.4.18 kernel. The host has an older version only because one of the unmodified rootkits cannot run on newer version of kernel. Both kernels are slightly modified. In SecCore, we added a set of functions to probe PCI device and setup shared memory. These routines discover the host computer, access the PCI configuration registers, configure the software-mapped memory address, request an I/O region and then enable DMA by using PCI-BIOS APIs. Since the configured PCI address is a physical address, it must be `ioremap`ped to a kernel virtual address so other software can reference it.

In section 6.1, we briefly show related information about PCI architecture since the SecCore heavily depends on it. Then in section 6.2 we present the data structure of a binary that is needed by an integrity builder routine to find and compute an initial integrity. The software builder should become a system call service in operating system so that end users can use it. In section 6.3 and 6.4, we describe the newly added routines in host kernel, SecISR and checkers, respectively .

6.1 PCI System Architecture

In general, most of the PCI devices can at least supports three types of resource sharing - PCI-shared memory, interrupt pin and bus mastering. The PCI bus is the most common architecture in consumer computers that allow different peripheral devices attaching to a platform. The PCI specification covers the electrical characteristics, bus timing and protocols. One of its important features is that it has a slot based configuration space so that each PCI device has different address decoder. The configuration is done by BIOS or OS drivers, so each device is allocated a mutual-exclusive bus address space. Such assigned I/O space is called *PCI-shared memory*, *memory-mapped IO*, or *IO memory*. This PCI-shared memory is physically located inside the physical device but the host can access it just as if it accesses the main memory. For example, when the CPU issues a read from PCI-shared memory region, the data are actually located inside the PCI device. Under Intel x86 architecture, the system primary PCI expansion bus is physically attached to ICH (IO Controller Hub) and system memory is attached to MCH (Memory Controller Hub). These two specialized ICH and MCH chip are also known as chipset. The chipset connecting PCI devices decides the memory range allowed to CPU. Many Intel's chipsets allow up to 64kb+3 bytes to be addressed within I/O memory space.

In our case, the chipset sees the SecCore as a regular PCI device. The host CPU will assign PCI-shared memory space to it. Similarly, The SecCore also sees the host computer as a PCI device and therefore it can access the host memory. Using the PCI-shared memory in this matter is not a new technology. Several of commercial products have

merged the PCI-shared memory from each PCI SBCs (Single Board Computer) into a tightly coupled cluster. In their cluster, the inter-node communications are going through PCI-shared memory across PCI bus.

The SecCore needs a mechanism to communicate with host CPU. Using hardware interrupt is the most common way to notify the host CPU to execute the corresponding interrupt handler. To share the interrupt line and number, each device including the SecCore must be assigned a mutual exclusive interrupt number to avoid conflict.

Bus mastering feature is optional. This technique enables the bus controller to communicate directly with other devices without CPU's attention. It allows a device to be a master, so it can drive data bus and directly read from/write to memory bank and control signals. DMA is a simple form of bus mastering where the I/O device is set up by CPU to access memory block and then signal CPU when I/O is completed. If the host system supports the bus mastering or DMA, the SecCore can directly and quickly transfer the PCI-shared memory block on host. However, it is not required, because the SecCore only needs to access a small memory block, it will not cause a lot of performance impact.

6.2 Software Integrity Builder

The ELF (Executable and Linkable Format) file is the most common executable format in Linux systems. There are three types of ELF object files - executable file, relocatable file and shared library. Each ELF file is composed of one ELF header holding the roadmap of file structure, followed by program and section headers. The section headers focus on where various parts of the program are within the file, while the program headers describe where and how these parts are to be located in memory.

If no integrity is provided from vendor, an initial hash needs to be built right after the executable is generated or installed. We developed a routine, called *software integrity builder* (or simply *builder*) to scan the ELF headers of a given software, locate its code segment, and then compute the MD5 hash. The checksum function is taken from Linux utility `md5sum.c` and specialized for ELF format. The basic logic of building user-level software and kernel are almost identical. An application has exact one identity, but a kernel is split into multiple integrities which we will discuss in later section **??**. The builder must call the SecCore to get the integrity data signed before storing them in a file. At this implementation, the builder uses host memory indicating a service request and also get results from there instead of directly writing the command register and reading the status register in SecCore.

We briefly show how an executable is generated and its internal format, because some of the information are related. In general, all programs are linked to `/lib/crt0.o` library that inserts a real entry point `_start`, initialized `.data`, and stack points. After compilation, the linker program `ld` combines a number of archieve files, and relocates data and symbol references. Alternatively, the `ld` accepts linker scripts to overwrite the default VMA (Virtual Memory Address) or LMA (Load Memory Address). The builder must locate the VMA, hash the contents and store the integrity along with its offset, code length and name for checkers to use.

At execution, the `sys_exec` is called by all `exec` wrappers. It uses few service routines to allocate page frames, prepare data structure, get `dentry`, file and `inode` object, copy arguments and environment variables, and scan the format to apply corresponding method. In ELF format, `load_binary` method is called to invoke `do_mmap` function to create a new memory region that maps text segment of the executable file. The initial linear address of the code by default starts from some default offset `0x08048000` in application. The kernel's linker script `linux/arch/i386/vmlinux.lds` set the kernel code start from `0xc0100000` by default. These data are essential for the checkers to locate the software image in virtual memory. Then the virtual address needs one more translation to become a final physical address. Finally, `load_binary` sets the values such as `start_stack`, `brk`, `start_code` of the process's memory descriptor, and invokes `start_thread` macro to modify registers so that `eip` will point to the entry point of the program interpreter and `esp` point to the new user stack, respectively.

Figure 3 shows a static memory instance of an executable and in-memory image. The left side shows an ELF executable processed by builder while generating an initial integrity. The right side shows an simplified running software image in memory that will be attested by software checker. During the attestation, the checker verifies the signature, uses the data to locate its `.text` and compare its integrity.

6.3 SecISR Routine

Rather than inventing a new interrupt vector, we embed the SecISR inside the system timer ISR (Interrupt Service Routine). The modified timer ISR and its routines are shown in Figure 4. An IDT (Interrupt Descriptor Table) is an array of descriptor that is used to associate interrupt and exceptions. Each IDT entry stores an interrupt handler's address. The IDT is allowed to store consecutively anywhere in memory and the kernel uses a special register `idtr` to keep the base address. When an interrupt occurs, the CPU loads `idtr` and uses the interrupt vector to locate the entry point, and in this case, it is `IRQ0x00_interrupt`. Each interrupt vector has its own ISR, but they all share some common design and routines.

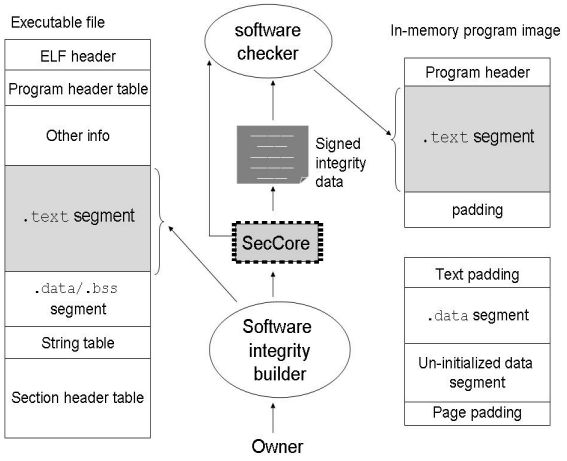

Figure 3. Building and Checking Software Integrity

Linux separated an ISR into top and bottom half. The top half is for the most time critical portion and bottom half is for non time critical portion that can be deferred. The kernel uses the generic facility `do_IRQ`, `handle_IRQ_event`, `tasklet_hi_schedule` and `do_softirq` to execute top half and schedule the bottom half to run. In the Figure, the last function in timer ISR is our `wake_up_checker`, that starts a number of software checkers.

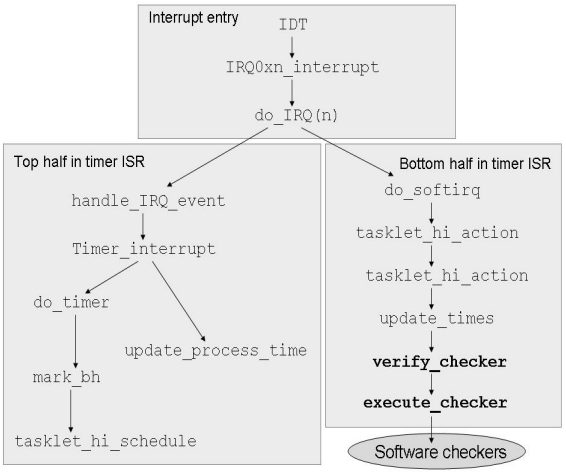

Figure 4. Modified timer interrupt handler with SecISR embedded

There are few interesting properties. First, the modified timer ISR can work with or without SecCore. If a system does not have the SecCore plugged in, it still runs as timer runs, and the SecISR becomes the root-of-trust. However, since the SecISR can be tampered by rootkits, not having SecCore is not secure. Second, `wake_up_checker` needs to verify the checkers before invoking them. The integrity data are stored in local file. It means `wake_up_checker` must perform additional file I/O operation. Third, `wake_up_checker` does not run the checkers in the timer interrupt context. Instead, we created a new thread for software checkers. Thus, the checking activity will not impact the performance in timer ISR and the checker can block on their own events.

6.4 Software Checkers

We implemented a number of software checkers in host kernel. The patched kernel can alternatively opt-in or opt-out by `make config`. Perhaps, it has no clear answer on how many checkers are optimal, but we developed four checkers based on their particularity and specialization. They are 1) system call checker, 2) module checker, 3) kernel checker, and 4) application checker.

The *syscall checker* is responsible for verifying all system calls and entries. System calls are the most popular target to attack by today's rootkits. The syscall checker first attests the `sys_call_table` array where address of system calls are kept. Next, it validates the integrity of the `.text` for each system service routine. Although these data are part of kernel, the advantage of separating it from other checking makes it easy to identify the exact compromised syscall entry.

The *module checker* is a little bit special from others. Kernel modules are linked into the kernel by executing the `insmod` user program. Linking a module requires a user process to interact with the kernel service back and forth few times. The `create_module` replaces all external and global symbols with corresponding logical address. Use of loadable modules is both a convenience and a serious vulnerability. The fundamental challenge is that modules do not have an initial integrity like others. Therefore, in addition to attestations, the module checker needs to perform more work. First, when a module is inserted, the checker needs to compute its integrity and save it in a list. We modified the `sys_init_module` and `sys_delete_module`, so they insert and remove an integrity as a module is loaded and unloaded. These data are maintained in a linked list and used by module checker. Second, in case of deletion, the checker is called to remove an entry from the list. It inspects the caller's return address in the stack frame to make sure only the `sys_delete_module` can do so. This prevents some rootkits from directly modifying the kernel `module_list` data to hide themselves.

The *kernel checker* will verify kernel code segment in `.text` and `.text.init`. We skip about first five pages in `.text.init` section where it contains data such as

`idt`, `gdt`, `swapper_pg_dir` and so on.

The *application checker* runs in a kernel thread. It walks thru the kernel process table, and if a registered application is found to be checked, it indexes the process's VMAs from `task_struct->mm->mmap` followed by another round of translation from VMA to kernel virtual address using the page table entry `pgd`. The rest of checking operations are identical to other checkers.

7 Evaluation and Performance

Our evaluation consists of four parts. First, we test the effectiveness and correctness of our system against some real-world common rootkits. Second, we test it against our own rootkit utility. Third, we evaluate the software updates. And the forth part measures the performance overhead on host.

To evaluate our model can detect the rootkits attacks, we took a few well known real-world kernel rootkits. These rootkits are obtained from public source domain and all are not modified. *Adore* and *Adore-ng* come by kernel module that they intercept the execution flow by altering system call table and virtual file system. *SuckIT* presents a different attack, it is loaded through `/dev/kmem` by writing to the memory special file. We install those rootkits one at a time, our checkers could successfully detect when Adore replaced system call table, and when SuckIT rewrote the raw memory. The checker also discovered when Adore-ng removed itself from module list, however, the checker did not recognize Adore-NG illegally altered the VFS function pointers. We will discuss this limitation in section 8.

Beyond the known rootkits, we developed our own synthetic rootkit that can arbitrarily replace the memory contents in any location. Because our detection scheme is not known to real-world rootkits, thus we instrumented our rootkit utility to randomly modify the `.text` area of given user processes, kernel routines, checkers and even SecISR. The binary rewrite actions were all successfully detected by different checkers. However, we currently do not have a well-designed rootkit that can rewrite the checkers and SecISR without crashing them. A summary of above attacks and outcomes are depicted in Table 1.

To test a legal software update, we modified the SecISR and a kernel routine and re-computed their integrities. The new values were displayed in SecIO and the we commanded the SecCore to sign it. After that, the SecCore and SecISR could successfully adapt the updates and start using these new values.

Finally, in performance tests, we configured our system to execute the checking about every five seconds. No measurement was taken on SecCore side because its operations do not interfere with the host CPU. we measured that the performance impact on host computer is really small. During several measurements, the checkers completed in one to five jiffies, where one jiffy is typically about ten microseconds. In case of large applications such as database, there would be some extra cost for kernel to page-fault all missing pages and bring them in due to user programs are page-on-demand based. In a system where memory usage is high, additional swap-in and swap-out are required, that could certainly worsen the system performance. However, it does not happen to kernel checking because kernel itself will never be swapped out. Running the checking every five seconds is simply a random choice because there are no known rootkits or viruses that repeat loading and unloading themselves within every five seconds.

8 Open Issues and Limitations

There are two limitations in our model.

1. Our integrity check does not include the data section (`.bss` and `.data`). Some programs store pointers to function calls in data variables. Therefore, it is possible to alter data pointers without being detected, as we showed discussed in Adore-NG rootkit earlier. An example is the Linux VFS (Virtual File System) which binds a data pointer to proper operations associated with that file system at runtime by storing the pointer as data. We currently do not have a solution for this open problem. From the SecCore's point of view, it cannot attest the data section, since the data are dynamic and they are constantly changing. The operating system should have a mechanism to validate the VFS data pointer and execution flow, because an external device could never know how to properly handle them.

2. Our prototype implementation is tightly bound to Linux kernel and ELF format. Other operating systems and execution types are not supported. However, we believe it is feasible to migrate across platforms. Additionally, the builder and checker do not interpret the executable using standard BDF (Binary File Descriptor) library as most GNU utilities do.

We also have two open issues to discuss.

1. The first concern is about the social engineering. Despite of whatever robust security scheme is deployed, computers can still be very vulnerable if the owners voluntarily install malware or permit malicious update. Software vendors could provide better information about code signing, software identity, software certificate and so on. In software tamper resistance world, a solution must take users, software and hardware vendor all together alone with security mechanism.

Name	Target	Type	Source	Result
Adore 0.34	Syscall and table	Kernel	Loadable kernel module	Detected
Adore-NG 1.31	VFS	Kernel	Loadable kernel module	Detected
SuckIT 1.3b	Syscall handler	Kernel	Raw memory access	Detected
Our Utility	Any functions	Kernel/User	Raw memory access	Detected

Table 1. Common Rootkits and Detection Results

2. The second one is the ease-of-use concern. This is especially true in consumer domain. Usability is paramount. Our security mechanism automatically runs in background and is unobtrusive. Also the overhead cost is minimal. However, user interaction is really unavoidable in some cases such as recovery or updates. Computer owners should really be aware of the importance of this defense line and accept the extra human-in-loop inconvenience.

9 Conclusion

This paper has presented our hardware-software co-design for software integrity assurance. Attesting software integrity is a difficult but critical task, and it is the cornerstone in building a secure computing environment. Running on separate hardware, the SecCore and its verification functionality remains in place even when the host kernel is thoroughly compromised. The SecCore is time-driven, independent, and self-sufficient, and it activates other software checkers on a regular basis to build up a hierarchical trusted chain. The SecIO is exploited to interact with the owner for an authentic message. This out-of-band I/O device along with the SecCore provide a flexible and robust security solution on software integrity assurance.

We also demonstrate a prototype implementation of our design approach. The security hardware is simulated by PCI SBC device, and the software on SecCore and host computer are a patched Linux kernel. We added functionalities to support PCI interconnection, and we also developed routines to build an initial software integrity and compare it with the running software in memory. Our prototype demonstrates both feasibility and efficiency with the hierarchical checking. The real rootkits attacks indicated that almost all intrusions are detected.

References

[1] "Computer virus prevalence survey, icsa labs 10th annual," http://www.icsa.net/icsa/docs/html/library/whitepapers/VPS2004.pdf.

[2] "F-secure's data security wrap-up for january to june 2006," http://www.f-secure.com/2006/1.

[3] "Mcafee virtual criminology report," http://www.mcafee.com/us/local_content/misc/mcafee_na_virtual_criminology_report.pdf.

[4] Trend Micro Inc., "Vulnerability exploits break records," *White paper*, 2006.

[5] "Sony, rootkits, and digital rights management gone too far," http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rghts.htm.

[6] McAfee Inc., "Rootkits, part 1 of 3: The growing threat," *White paper*, 2006.

[7] F. Cohen, *Computer Viruses*, Ph.D. thesis, University of Southern California, 1986.

[8] "Processors get hardened - security concerns are mandating the enhancement of on-chip protection," 2004, http://www.trustedcomputinggroup.org/home/409153.pdf.

[9] "Lagrande technology for safer computing," http://www.intel.com/technology/security.

[10] "Microsoft next-generation secure computing base - technical faq.," http://www.microsoft.com/technet/archive/security/news/ngscb.mspx.

[11] "Trusted computing group," http://www.trustedcomputinggroup.org/.

[12] J. Molina N.L. Petroni, T. Fraser and W. Arbaugh, "Copilot: A coprocessor-based kernel runtime integrity monitor," *In 13th USENIX Security Symposium*, Aug. 2004.

[13] K. Thompson, "Reflections on trusting trust," *Communications ACM, Vol 27*, Aug. 1984.

[14] M. Gasser S. Ames and R. Schell, "Security kernel design and implementation: An introduction," *IEEE Computer*, 1983.

[15] K. Wika and J. Knight, "A safety kernel architecture," Tech. Rep., 1994.

[16] R. Farrow, "The kernelize secure operating system (ksos)," *USENIX and SAFE magazine, Inside: Security*, Dec. 2002.

[17] et al. S. King, P. Chen, "Subvirt: Implementing malware with virtual machines," *In 2006 IEEE Symposium on Security and Privacy*, May 2006.

[18] B. Yee, *Using Secure Coprocessors*, Ph.D. thesis, Carnegie Mellon University, 1994.

[19] "Ibm pci cryptographic coprocessor," http://www-03.ibm.com/security/cryptocards/overproduct.shtml.

[20] "Intel pentium iii processor id," http://www.intel.com/support/processors/pentiumIII.

[21] S.R. Hedberg, "Hp's international cryptography framework: Compromise or threat," *IEEE Computer 30*, 1997.

[22] "Sun microsystem," http://sun.com.

[23] D.J. Farber W.A. Arbaugh and J.M. Smith, "Secure and reliable bootstrap architecture," *In IEEE Symposium on Security and Privacy*, May 1997.

[24] W. Koehler D. M. Alexander, "Trusted computing: From theory to practice in the real world," *Information Security Solutions Europe 2004 Conference*, 2004.

[25] "Press release - microsoft palladium: A business overview," http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp.

[26] "Not tcg debate," http://www.notcpa.org/.

[27] "Against tcg," http://againsttcpa.com/.