Journal of Computer Security 00 (2008) 1–29
IOS Press

Coprocessor-based hierarchical trust management for software integrity and digital identity protection

Lifu Wang and Partha Dasgupta

Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, USA
E-mails: {Lifu.Wang, partha}@asu.edu; Tel.: +480 965 5583; Fax: +480 965 2751

Malware and rootkits are serious security threats. They can be designed to be resistant to anti-virus and security software and even remain totally undetectable. This paper describes a hierarchical trust management scheme, where the root of trust is in a non-tamperable hardware co-processor on a PCI bus. The security device checks a part of the OS kernel for integrity, which in turn checks other parts until we ensure the entire system is free of rootkits. The checker can be extended to encompass all applications and anti-virus software. Our system can detect any illegal modifications to kernel, loadable kernel modules and user applications. It also provides a secure communication line for user interaction to manage legal software updates. Moreover, this device can securely perform user authentication and protect digital identity against identity theft. Our tests show that we can correctly detect different real-world and synthetic rootkits even though the host kernel is compromised.

Keywords: Computer security, software integrity, digital identity

1. Introduction

Malware have been a serious security problem over past decades. Attackers use malware written to exploit software weaknesses, eavesdrop on unprotected network traffic, steal passwords, and to penetrate computer systems connected to the Internet. Once access is gained, attackers install malware on a victim's machine to spread viruses or worms, steal sensitive information, make the host into a zombie, botnet or whatsoever to serve their malicious purpose. In 2006, F-Secure [1] reported that over 185,000 viruses are known. Another survey [2] found that the virus infection rate has increased 116% per month and the caused damages increased for ten consecutive years.

Amongst malware, rootkits are the most dangerous because not only are they very difficult to detect but also they are often used to disable other security software. A rootkit is usually installed by patching the kernel so that the rootkit runs inside the operating system. Thus a rootkit is very powerful and useful in hiding other malicious programs or intruder's activity. For example, to make viruses invisible, a rootkit can disable any anti-virus process or forge the virus definition database. To hide the existence of an intruder, a rootkit can patch related kernel routines to trick system utilities

0926-227X/08/$17.00 © 2008 – IOS Press and the authors. All rights reserved

UNCORRECTED PROOF

such as `ps` or `netstat` to not show certain login sessions, processes, open ports, communication or user behavior. In essence, rootkits are the root cause of many other security problems including virus infection or identity thefts, and therefore solving the rootkit problem deserves a higher priority than other security problems. McAfee found [3] that the frequency of rootkit attacks has been beyond all other forms of malware, and the number has increased from 300 per month in 2004 to 2,000 per month in 2006. Today, not only are virus authors embedding this stealth technology in their malware, but commercial software [4,5] were also found to deploy rootkits.

Since rootkits and viruses can compromise any software, attestation of code authenticity becomes an important issue. To ensure a rootkit free computing environment, all software including OS and applications must be constantly monitored to ensure their instructions are not illegally modified. Maintaining code integrity is paramount; otherwise, software cannot be expected and trusted to perform as intended.

In this paper, we present our integrated hardware/software design. First and foremost, all software should have a pre-computed "good" integrity, also known as *baseline integrity*. The baseline integrity is either locally computed right after software installation, or it is a digital signature with an associated certificate from the vendor. A valid certified signature indicates that the software is truly from the trusted vendor and the software has not been altered since it was signed. With this baseline integrity, the *checker* or *verifier*, can use it to compare with the integrity of current running software inside physical memory. Hence, any illegal modification by rootkits or viruses can be detected due to mismatched integrity. The verifier must be implemented and installed in a secure way to ensure that no malware can compromise it, disable it, or spoof the results it generates.

In addition to our security hardware/software, human interactions play an important role in our design for two critical decision making cases. First, when the checker detects an installed rootkit, the user must be securely notified about the incident. Second, when a software update occurs and no valid signature is provided, a user confirmation is enforced since there is no way to determine the update is driven by user or attacker/malware. Therefore, it is important to provide a trusted out-of-band communication path that guarantees all input and output messages to/from user are un-tamperable or un-forgeable by malicious I/O drivers or attackers.

Within our design, the lowest level is a PKI (Public Key Infrastructure) device added into a host computer. This device is composed of two components – the "Security Core", called **SecCore** and "Secure I/O", called **SecIO**. The SecCore is connected to the host via PCI bus so it can access the host memory and execute its trusted software to verify integrity of a critical section in the running host kernel for integrity, and then interrupt the host CPU to execute a specific interrupt routine that will initiate a number of checkers. These checkers will verify the kernel, modules and applications in a hierarchical structure. Once the chain is built up, i.e. software are verified one after the other, the host computer is claimed secure. The SecIO is a simple I/O device that is directly connected to the SecCore. It contains a numeric

UNCORRECTED PROOF

keypad input and a small LCD display output. The keypad is used to enter special commands, while the display is driven only by the SecCore to show a text message. The SecIO facility ensures that I/Os between the SecCore and user are not disrupted, because the host CPU has no control over this hardware. The SecCore is also a storage for safe-keeping sensitive information such as digital identity and furthermore, by running PKI protocol with SecCore/SecIO, we can secure a user authentication and financial transaction against identity theft.

Our solution bears some similarity to TPM and Copilot. A TPM (Trusted Platform Module) [6] chip with a LT (LaGrande Technology) [7] motherboard and NGSCB (Next Generation Secure Computing Base) [8] secure operating system is a security solution from TCG (Trusted Computer Group) [9]. The TPM may be capable of secure bootstrap but subsequent deployment of malware can go undetected. The TPM also has no secure I/O, which is a severe shortcoming. The Copilot [10] is a hardware coprocessor that constantly monitors the host kernel integrity. It cannot handle dynamic kernel modules and user-level applications and it does not have a mechanism for a kernel patch. Our hierarchical checkers can attest the kernel, modules and applications. In addition, our SecIO is a much needed mechanism for ensuring secure kernel updates.

The remainder of this paper is structured as follows. Section 2 describes background and a typical attack model. Section 3 states the problem, Section 4 defines the threat model and Section 5 describes related work. Section 6 discusses our solution and system design. Sections 8 to 9 present our implementation and simulation. Section 10 discusses the limitations and open issues. Finally, Section 11 summarizes our work.

2. Background in malware threat and attacks

The prevalence and availability of malware makes hacking a computer easier than before. We summarize a few malware trends from various reports [11–15] as follows.

1. Combination – Different types of malicious code are being merged into multi-purposed hybrid versions. A single virus becomes more malicious when it contains rootkits, viruses, worms, spyware and key logger.
2. Automation – The software-based attack is favored by all intruders because of its low cost and high performance. Software can automatically search for potential victims and crack into unprotected machines. With these tools available on-line for every one, an attacker requires few computer skills.
3. Complexity – Malware technology is getting more sophisticated every day, but the development time is getting shorter. New malicious logic is more difficult to identify or discover through current anti-virus software or intrusion detection systems.

UNCORRECTED PROOF

4. Competition – As software bugs are identified, hackers compete with each other to write tools to promptly exploit the security hole. These tools enable attackers to intrude other computer systems before their owners can patch their computers.
5. Marketing – People are now selling their malicious programs for profit. Today, an estimated 85% of malware is written purely for profit. Cyber criminals purchase and use these malware for identity theft and financial gain. The market is becoming even more commercialized and more crowded as legal data mining companies and illegal organized crimes are both evolving.

We briefly identify those frequently used terminologies throughout the rest of the paper. A *vulnerability* is a software weakness or flaw that can be exploited. Bugs are the most basic form of software vulnerabilities. The large number of vulnerabilities is in fact due to incredible software complexity and poor programming skills. According to CERT, the reported vulnerabilities have, on average, doubled every year for 10 years. As software continues to become more sophisticated, the number of vulnerabilities will never decrease.

An *exploit* is a piece of code, a chunk of data or a sequence of commands that takes advantage of vulnerable software in order to cause unintended or unanticipated behavior to occur. Most exploits are designed to gain a superuser-level access to a computer system. Sometimes several exploits are used first to gain lower level access then to escalate privileges repeatedly until root access is reached. A number of security incidents exploit a web browser's vulnerability. Today, after a vulnerability is discovered, a corresponding exploitation tool will often be ready within a few days, as opposed to 11 months in 2001 [12].

Malware, or *malicious software*, is software used by attackers and is designed to infiltrate or damage a computer without its owner's consent. Traditionally, based on their purpose, malware can be classified into different categories. Viruses and worms are the best known type of malware and they are infectious for the manner in which they spread themselves. Trojan horses, rootkits and backdoor are concealment malware; they are disguised as something innocuous or desirable to install so that users will install them without knowing what they will do. Spyware, botnet, logger and dialer are generally written with financial profit in mind. However, malware are becoming hybridized and more harmful, so the classification is no longer that clear and important.

The most popular software vulnerability is a *buffer overflow*, and the most popular exploit is called *buffer overflow attack*. A buffer overflow is an anomalous condition where a program writes data beyond the end of a buffer allocated in memory stack or heap. It is usually due to improper use of memory, unsafe programming languages such as C/C++, when a process inputs more data into a buffer than it was intended to hold. If the amount of data written into a buffer exceeds the buffer size, the additional data will be over-written into adjacent areas. An attacker can overflow the buffer, feeding specially crafted input content designed to trigger other specific actions such as opening a trapdoor, installing a virus or starting a shell script.

UNCORRECTED PROOF

A *rootkit* is a special kind of malicious code commonly used to compromise the OS or system utilities. A user-level rootkit replaces the application binary or modifies the behavior of an existing application using hooks, injected code, etc. A kernel-level rootkit adds malicious code into the OS or modifies a portion of some kernel functions, which can be accomplished by a variety of means such as OS vulnerability, loadable modules, patches or system tables. Rootkits allow intruders to maintain a root privilege on a system so they can hide logins, processes, and files, intercept data from I/O devices, and disable malware detectors.

Rootkits are becoming more sophisticated, but it is not difficult to adapt and use them. In fact, the technology has become so mature that even commercial software companies are now using it. Sony BMG's XCP software was found [4] to employ rootkits to hide their digital right management files and processes so the user could not disable it or make illegal copies of their music. Even an anti-virus company was found [5] to install rootkits to hide a directory for their virus detection purposes. In the past three years, the use of rootkits technology has grown by more than 600% [13]. A study [16] showed that the frequency of rootkit attacks has grown beyond all other forms of malware. The same study found that injecting a rootkit is easier than before.

A typical step-by-step intrusion may involve four phases: (1) search open ports on a network for a target by using port scanner software, (2) exploit an operating system or application vulnerability such as buffer overflow to sneak into the target machine, (3) once access is gained, place viruses or the like for any malicious purpose, and (4) install rootkits to disable any security software and hide installed malware. Neither firewall or anti-virus tools can detect or prevent such attacks.

3. Problem statement and challenges

In this paper, we define the problem as *how to guarantee that an arbitrary program can execute untampered by malware on an untrusted host computer*. Software tamper-resistance has been an open research challenge for a long time. It is a foundation of computer security and there is no way a host computer can be secure, if its running software is not authentic. A code execution can be maliciously tampered with in many ways, such as (1) modify the code before invoking it, (2) inject or execute alternate code, or (3) modify the execution state in memory or registers when the code is running. Any of them can result in compromised integrity. In this paper, we assume an adversary can completely subvert the host computers and gain root privilege via a network. However, we assume an adversary does not have any physical access to host computers. This means that the external security hardware and software are inaccessible to attackers since they are not under host computer control.

Current file integrity checkers, like `tripwire` or `mk5sum`, compute an initial checksum of an executable file so that at a later time they can use it to compare with

UNCORRECTED PROOF

the current checksum. These utilities are designed to detect data corruption or transmission errors in file but not in memory. Checking static file is simply not enough, because as mentioned earlier even a binary at the beginning of its execution may be verifiable, but while running, its execution flow can be redirected to externally inject malicious code. Rootkit detection is more than computing and comparing two checksums. Furthermore, if the checker is not deployed correctly, it could become useless. First, because the checker is another set of software, rootkits can always find it and disable it. This leads to an endless arms race – the checker will detect and kill rootkits or vice versa. Second, rootkits can tamper with the baseline integrity on storage. Using cryptography functions to encrypt data does not solve the problem either, because rootkits can recover the cryptographic key and then the data. Third, rootkits can do binary-rewrite attacks to compromise the checker.

Another popular approach is anti-virus software that can identify malicious logic in a program, so malware will be prevented in the first place. Unfortunately, there is no perfect way to determine whether any given software is malicious or not. Cohen [17] has proven that mathematically perfect detections of unknown viruses is equivalent to solving the *halting problem*. That means no arbitrary algorithm can look at other arbitrary programs, including itself and determine either "malicious logic present" or "no malicious logic present" in finite time, that is, the problem is *Turing Undecidable*. While we cannot write programs to detect all unknown viruses, it turns out that even when we have a sample of the virus in hand and have analyzed it completely, we cannot write a program that detects just that particular virus without false positives [18].

Anti-virus software is the most widely used security software today. It had some success in limiting known viruses but all virus scanners suffer from two major problems: (1) they cannot detect new viruses and (2) they can be disabled by rootkits. Anti-virus software uses heuristic workaround to detect viruses or malware. In a static detection scheme, it goes through each file and looks for malicious fingerprints or patterns. Because anti-virus software must keep all virus fingerprints, as the size of its database dramatically increases, this scheme becomes very inefficient. In a dynamic heuristic detection scheme, it observes and analyzes suspicious behaviors, but this scheme often leads to an erroneous result. A false positive error may be just annoying, but a false negative error is dangerous.

A firewall is a network security device, hardware and/or software based, to keep out unwanted traffic or unauthorized access to a private network. Firewalls can be configured to permit, deny or proxy data connections to control traffic between computer networks with different zones of trust. Unlike anti-virus, firewalls work in the background at the device (link layer) level to protect a system. Using a firewall to limit access can prevent a lot of random or drive-by attacks. However, a firewall cannot protect vulnerable software like a web browser from being exploited. Also a firewall is not for every one; a proper configuration demands administrator skills and understanding of network protocols and computer security, which means a small mistake can render a firewall worthless. Home users are often found to remove their personal firewalls because of their annoying messages and inconvenience.

UNCORRECTED PROOF

4. 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 the *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) defend against 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 Thompson [19]. He suggested no amount of source level verification or scrutiny will protect us from malicious code. We call it the *Thompson Threat Model*. Since it is impractical for anyone to create the entire software stack, assembler, compiler, kernel and libraries from scratch, this threat model is too strict to have any feasible solution.

We define our threat model as the *Viral Threat Model*. It is a weaker 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 initial software from the vendor. In the Viral Threat Model, attackers crack into a host and install malware by various ways to compromise the system. Our goal is to detect such infractions.

Computer users fear using the Internet because of the large amount of warnings about security issues. They have no confidence in sending credential data over the Internet, but they fear for the wrong reasons. It is malicious software inside host computers, rather than the network, making the ever-present security threats worse. 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. In order to give consumers a higher degree of confidence in using their computers, our goal is to *keep consumer computers safe from malicious code execution under 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 digital right management and resource stealing issues either.

5. Related work

We summarize the prior work in rootkits detection and prevention into software based, OS based, hardware (trusted computing) based and VMM based.

UNCORRECTED PROOF

5.1. Software-based

A software rootkit detector, such as `chkrootkit`, `rkhunter` and `RootkitRevealer`, must be able to check the following areas in kernel, (1) `.text` section, (2) interrupt table, (3) system call table and (4) installed modules. There are different kinds of software checkers designed to detect a rootkit. In general, a software detector can be deployed in three ways.

In the first type, a detector is executed as a user program, but has a root privilege so it can check the weak points of the system. However, the detector itself is vulnerable and easy to be defeated. In the second type, a detector adds itself into the kernel by writing a `/proc/kmem` interface. It is required to carefully process the system symbol table and its data structure and the techniques are complicated and hence prone to errors. The `/proc/kmem` becomes a battle ground between detectors and rootkits, which can detect or disable each other. In the third type, a detector is dynamically inserted into a kernel as a loadable module. Since rootkits can attack the kernel, similarly they can attack any kernel module. Additionally, modules are not currently supported by all operating systems. It is apparent that no software detector is robust enough. Rootkit detection, in order to be foolproof, really should not be controllable by the OS, unless there is a trusted kernel.

5.2. Security OS-based

The most conservative approach is to directly improve the reliability of operating systems. Many of early researchers addressed building a security kernel [20–22]. A security kernel implements the basic security procedures to control the system resource, prevent unauthorized access, protect from modification and be verifiable. If there exists a security kernel, the rest of software can be securely compiled, executed and verified. The problem is that a reliable and secure OS did not exist in the past and probably will not exist in the future [23], because operating systems are not only too big, but their fault isolation is very poor.

First, all of today's commodity OSs are becoming really too large and complicated with a rich set of features. For example, the Linux 2.4 kernel has more than 2.5 million lines of code and the Windows XP kernel is more than twice as large. One study [24] showed that on average code contains between 6 and 16 bugs per 1,000 lines in an executable [24]. Another study [25] found the fault density is about 2 to 75 bugs per 1,000 lines in operating systems. About 70% of an OS consists of device drivers, which have error rates three to seven times higher than ordinary kernel code [26], so the bug count in any kernel is probably grossly underestimated than those cited above. Finding all OS bugs and fixing them all are simply not feasible and not realistic.

Second, most operating systems are monolithic kernel based and contain thousands of procedures linked together as a single binary. Without fault isolation in a kernel, if a rootkit infects one kernel function, there is no way to keep it from rapidly

UNCORRECTED PROOF

spreading to others and taking control of the entire machine. One research effort [27] makes the device driver, the most vulnerable part in commodity operating systems, less dangerous. The concept is to protect the kernel against malicious drivers by isolating them into a protection domain. The work focused on the device driver, rather than stopping or detecting malicious kernel routines, which may be directly modified by rootkits or installed from a patch.

Another approach is the secure and reliable bootstrap architecture [28]. In this approach, with the AEGIS platform using modified firmware, a cryptographic hash value of the boot process is checked, beginning at power-on and continuing until the transfer of control to the operating system. This is compared with a stored digital signature associated with each component to guarantee that the bootstrap is in a secure state. For example, the BIOS verifies a public-key signature in disk's boot sector for authenticity; the boot sector then verifies the signature of the OS bootstrap code, which likewise verifies the privileged processes and drivers. This scheme has a few weaknesses. First, the BIOS in most consumer computers is writable, which means that the malware can update the BIOS. One solution is to store the BIOS on a ROM. However, a ROM-based approach is inflexible and prevents BIOS updates. Second, the use of digital signatures introduces a key management problem that is amplified by the requirement to store the initial public key in a safe storage. Furthermore, rootkits can rewrite running operating systems, for example modifying the system call table and routines instead of replacing the kernel image on disk, and such attacks cannot be solved by AEGIS alone.

5.3. Trusted computing-based detection

In recent years, security hardware has become mainstream. It was eventually realized that software-only security is not adequate to stop malware [29]. Analysts from IDC forecast that by 2007, 80% of computers will be equipped with hardware-based security devices rather than security software.

The Trusted Computing Group (TCG) [9] formed in 2003 merging major vendors and their work into an alliance. Its goal is to develop, define, and promote open, vendor-neutral industry specifications for trusted computing, of which all are required for a fully trusted system across multiple platforms and operation systems. The TCG has published two key building blocks: the Trusted Platform Module (TPM) [6] hardware and TPM software stack (TSS) specification. Based on the specifications, many TPM-enabled boards available today, such as Intel's LT (LaGrande Technology) [7] platform with a TPM chip integrated into the chipset. The most controversial features in TPM are (1) remote attestation, (2) binding and (3) sealing. *Remote attestation* creates an unforgeable summary of the hardware, boot, and host O/S configuration of a computer, allowing a third party, such as a digital music store, to verify that the software has not been changed. *Sealing* encrypts data in such a way that it may be decrypted only in the exact same state, which means it may be decrypted only on the computer it was encrypted running the same software. *Binding*

UNCORRECTED PROOF

encrypts data using the TPM endorsement key, a unique RSA key stored in TPM during its production. The future of the TPM based computing is not clear due to its complexity and the large amount of critics [30] pointing out that the primary function of the TPM is anti-consumer (i.e. DRM). The remote attestation feature is seen as a potential threat to privacy by many, while the binding and sealing feature are often seen as a herald to digital rights management systems of unprecedented restrictiveness. Because of the political debate, the TPM is currently a supported option that is generally turned off and is controllable via the OS. Hence, if the OS can turn the TPM off, so can malware in order to dodge detection.

The NGSCB (Next Generation Secure Computing Base) [8,31] from Microsoft is a software architecture on top of a TPM embedded platform. In NGSCB, they implemented a new security kernel, called Nexus, and a number of NGSCB-enabled trusted software, called NCAs (Nexus Computing Agents). The NGSCB relies on TPM's secure storage to perform cryptographic operations and TPM's curtained memory to isolate access. Although Microsoft claims the NGSCB can enable the trusted computing to increase the security for end users, critics assert that the technology will result in vendor lock-in software. Our work is different from NGSCB/TPM technology in different ways. First, our solution does not require re-designing a new VMM-like trusted security microkernel as NGSCB does. Second, applications are not classified and separated into trusted and untrusted lands as NGSCB does. And third, our design re-uses existing hardware/software as much as possible, so it is less expensive and simpler.

The Copilot [10] is another hardware coprocessor-based solution that supports a kernel integrity monitor for commodity systems. It can detect malicious modification even if a host kernel is thoroughly compromised. The Copilot contains an external PCI card and software running on the card. Its goal is to remain as independent of the potentially subverted operating system as possible. To do this, the PCI card has its own CPU and uses DMA (Direct Memory Access) to scan the physical memory of the computer looking for rootkit behavior. The PCI board also has its own network interface to communicate in a secure fashion to an administrative component. This mechanism is more hardware-oriented and thus has some shortcomings. First, the Copilot cannot monitor the integrity of dynamic loadable kernel modules and applications, since it does not have any knowledge about where they are loaded by the operating system. Also the PCI card can only access the host memory but not host CPU registers. Second, the Copilot cannot distinguish whether an update is user-driven or rootkit-driven, so it cannot handle software updates. Our approach is more hardware-software balanced, and therefore we can verify both static kernel and dynamic modules/applications at run time. In addition, our approach allows a user to manage his legal software updates.

5.4. VMM-based detection

A virtual machine monitor (VMM) technique [32] adds a layer of software to emulate a computer hardware such that one hardware can be partitioned into multiple

UNCORRECTED PROOF

isolated instances. A number of research projects [33–35] built a trusted VMM for intrusion and rootkits detection.

The VMM-based SM (Security Manager) [36] is a small isolated software module, running inside a privileged VM on top of VMM. The SM can run security protocols and perform several run-time checks against the operating system. Since the VMM already provides the ability of directly inspecting software as well as hardware states of the guest OSs, the SM can retain the maximum visibility of guests. When the SM needs to be able to securely communicate with a human in order to perform a variety of notifications and obtain human input, it uses a different secure I/O channel so that rootkits cannot modify the displayed message or user responses.

6. Our approach and system design

Our design is rooted in the SecCore, which is embedded software running on untamperable hardware, that is not part of the CPU of the host machine. The SecCore checks the integrity of a small module of kernel code, that is placed inside the OS kernel as an interrupt handler (called the SecISR). After checking the SecISR, SecCore triggers it at specified intervals in time. From this root of trust, we add more checkers that perform integrity checks on the kernel and these checkers are checked for integrity by the SecISR software. In other words, the hierarchy starts from the root-of-trusted SecCore, expands up towards the interrupt handler and software checkers, and it continues to expand by verifying and adding more and more software into the chain.

First we introduce the SecCore and SecIO in Section 6.1. In Section 6.2, we present a mechanism of checking the host OS's integrity. The key point is similar to Copilot where a security device is external, independent, and uncontrollable by the host CPU. Then we describe the deficiency in this scheme and provide our solution in Section 6.3 with hierarchical checking. The solution places an interrupt handler and several other software checkers inside the host kernel. Once verified, this handler is triggered to invoke other checkers that will verify any running software. Our system has a flexible design to manage legal software updates, which will be discussed in Section 6.4. In Section 6.5, we show a number of attacks and analyses. And in Section 6.6, a new protocol is introduced to protect user's identity in authentication and transactions from identity theft.

6.1. SecCore and SecIO component

The SecCore is an embedded system that runs an operating system and software, and is self-activated 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 a separate tamper-resistant hardware inaccessible by host software. Figure 1 shows that the SecCore is a PCI device plugged into

UNCORRECTED PROOF

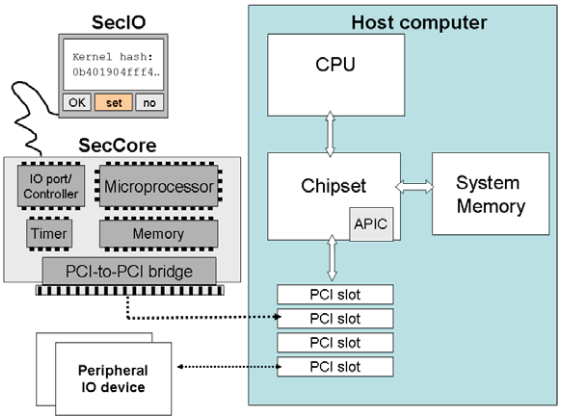

Fig. 1. The security components in a platform.

a PCI expansion slot. The SecCore shares the common basic features with crypto hardware and it has three additional requirements.

- 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 host system whenever necessary.

The SecCore consists of a microprocessor, memory, timer, PCI-to-PCI bridge and I/O controller. A low-end microprocessor can be used to implement the SecCore hardware. 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 to check host software and signals the host CPU. An I/O controller allows the SecIO to physically connect to the SecCore's I/O port. A PCI-to-PCI bridge provides an interface between the SecCore and the host chipset.

The SecCore runs a small operating system, only with basic kernel components and applications including 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. The 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) processors and network TOE (TCP/IP Offload Engine) chips. A large amount of crypto functions are well-defined mathematical computations, thus a FPGA is a good implementation choice. All functionality must be pre-programmed by the manufacturer. Even though, technically speaking, modifications or updates of the SecCore by users are possible, this ability should be disabled because this is a security risk.

UNCORRECTED PROOF

Communications between the host and SecCore are bi-directional. The host requests a service by writing to SecCore's command register and then reading its status register for results. These operations are typically done by `open`ing the SecCore device file followed by the `ioctl` system call. The SecCore asserts its interrupt pin to deliver a signal to the host CPU when it needs attention. All bulk data transfer goes through the DMAed PCI memory.

Figure 1 shows how the SecIO is connected to the SecCore. It has 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 a higher end TFT touch-screen. There are a lot of such devices available in today's market. Its keypad contains a set of special buttons, including "ok", "no" and "set". The SecIO is secure since the communication between the SecIO and the human user is not visible to, or tamperable by, the host operating system.

6.2. Checking kernel integrity

A kernel verification method is the computation of an initial integrity upon kernel initialization, which is then regularly re-checked to ensure no unauthorized modification. As stated earlier, this baseline integrity can be either locally computed or pre-computed by a software vendor. By integrity, we mean a hash (checksum, SHA or MD5 value) of the host kernel's code, known as `.text` segment. In practice, the SecCore only checks a part of, instead of the entire, kernel `.text` and the reason for this will be discussed later.

Checking a running kernel in memory is much more complicated than checking a static kernel in a file. We 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 is an ELF file. 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 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 the x86 platform. It means that 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, we can locate and check the kernel image in memory. Later we show why this is not an effective way for kernel integrity checkers.

Host memory access is done by mapping a PCI-shared host memory region 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 how the SecCore can address and verify the host kernel `.text`.

UNCORRECTED PROOF

Using SecCore to monitor the running kernel has two limitations – it cannot verify modules and it cannot verify user processes. First, the SecCore must know the pre-determined physical address and its size of `.text` segment, which 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 baseline integrity is. Second, most system services are implemented as root-privileged applications and they automatically start after the system is up. The SecCore has no way to know all running processes and page-mapping tables on the host, so it cannot verify any user application. It is impractical to make the simple software running on the SecCore hardware, aware of any dynamic host kernel data structure.

6.3. Hierarchical checking for software integrity

The checking mechanism is extended into a hierarchy such that all running software can be covered. The concept is straightforward and shown in Fig. 2. The SecCore is the "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 is called *SecISR*, is an interrupt service routine that will be executed by the host CPU as it receives an interrupt. At this point, the SecISR becomes "root-of-trust software". The SecISR is actually a starter routine – it validates and executes a *kernel checker* and *application checker*, which are kernel functions. The *kernel checker* and *application checker* then become the next trusted

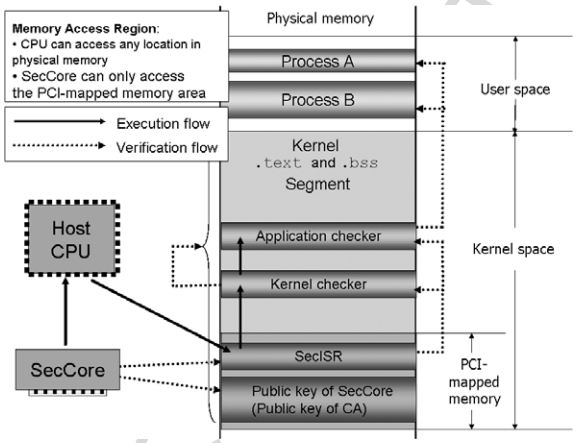

Fig. 2. Hierarchical checking.

UNCORRECTED PROOF

software in the hierarchy. The kernel checker verifies the entire OS and modules for integrity, while the application checker verifies any running process. Building up a trusted hierarchy always requires two operations – validation, if passed, followed by execution. In Fig. 2, a solid arrow represents an execution flow and a dotted arrow represents a validation flow.

There are several advantages in the hierarchical checking approach. First, the SecCore only knows the SecISR and nothing else. Since the SecISR is inside the kernel `.text`, its address, size and integrity can be pre-determined and kept by the SecCore. Second, because all checkers are software and performed by host CPU, this offloads the work from the powerless SecCore to the more powerful host CPU. Third, even though the SecCore shares host system memory, it does not necessarily mean that the SecCore is allowed to address any location out there. For example, a common allowance in PCI-mapped memory is usually less than 64 kB. It means a host will allow 64 kB of lower physical memory to be shared. This is too small for the SecCore to verify any user-level application which is usually loaded in a higher memory location. In our hierarchy checking, the only requirement is that the SecISR must be within the PCI-mapped memory region, while other checkers and applications can reside anywhere in memory.

The above scheme has one drawback – the baseline integrity must be available on the host, so the checkers can use it. The storage (file or memory) is vulnerable. The solution is to have the SecCore sign the data using its private key. Fig. 2 shows the public key of the SecCore stored in the kernel's `.bss` segment in lower memory. The kernel declares an un-initialized global variable and during boot, the kernel reads the public keys from a file into this variable. All checkers are programmed to read the public key from the known kernel variable so they can validate the signature of the baseline integrity before using it. And the public key is then verified by the SecCore so that no rootkits can fake a signature and public key.

6.4. Software update

Software updates, or patches, happen quite frequently nowadays, so our solution must be flexible enough to accept them. When an update occurs, the corresponding integrity must be maintained; otherwise, the checker will not have the right integrity at next verification. This process is essentially very similar to a flash (re)installation. However, the problem is, if an update does not have any valid signature, what should the SecCore do? From SecCore's point of view, it signs baseline integrity upon request, it does not know whether the request is malicious or not. For example, a computer user may decide to patch his anti-virus software and that will cause the system to re-compute a new baseline integrity and request for signature. At the next moment, a rootkit may rewrite that anti-virus software and make it look like another patch. If the later update is accepted, this compromised anti-virus software will never work as expected, and because the update is legitimate, no checker will detect anything

UNCORRECTED PROOF

unusual. In fact, no algorithm is as precise and effective as human decision to distinguish whether an update is user-initiated or is rootkit-initiated. However, having a human-in-the-loop may introduce another vulnerability, i.e. if all I/Os go through the host OS, the message could get intercepted and spoofed by a malicious driver. Even though the user says "no", the message could be altered into "yes".

The SecIO is brought to guarantee a genuine I/O message, because it provides an out-of-band secure communication channel. The exceptional and manual user interactions seem to cause extra workload, but the inconvenience does not have to be so drastic. The hierarchical checking can verify all software, but in practice there is no need to cover every arbitrary process like `a.out`. A better way is that at the application level, we only verify critical software such as anti-virus or `xinetd`. This means that only the SecISR, kernel, checkers, and critical applications are on the list. This way we reduce the number of human supervisors in software maintenance and let the anti-virus software continue to scan other programs. After all, our approach does not replace any anti-virus or security software, but co-exists and cooperates with these tools.

6.5. Attacks and analysis

There are different types of attacks to code integrity. The first type is an attack or set of attacks to an application's binary. If a virus infects an application, both the virus and infected application will be detected by the anti-virus tool. If a rootkit 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 the 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 the same time and tries to break the chain, it cannot be successful as long as the SecCore is functional.

The second type of attack compromises the initial integrity data. A rootkit cannot alter the SecCore's public key, because the key is inside the kernel `.bss` area and directly checked by the SecCore. A rootkit cannot tamper with any 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 being detected by SecCore. It is also impossible to make the checker read a fake public key elsewhere without modifying its binary.

The third type of attack makes the system accept some illegal updates. A rootkit cannot trick the SecCore into signing anything unless it is manually and securely authorized by the user. Finally, it is impossible for a rootkit to tamper with critical input and output messages between the SecIO and user.

UNCORRECTED PROOF

6.6. User identity protection in authentication and eCommerce

Even if our approach can securely detect compromised software, it cannot protect confidential information, for example, a user's digital identity, from being stolen. We show that the SecCore/SecIO along with the PKI protocol will provide a better solution in data protection and user identification.

Today's most common authentication method is based on shared secret, but this scheme leads to a severe identity fraud problem. For example, computers identify a user by username and password, and a financial transaction is secured by account number and PIN. It all depends on the ability of both parties, local and remote host computer, to keep the secret safe. No matter how carefully a password is selected, there is no confidence as to how it is possessed, processed and stored by a remote party. There is no known method that can securely uphold a shared secret on an untrusted host computer.

Since the discovery of PKI, it has been obvious that it is the only known way to provide a promising authentication method without a shared secret. During PKI user authentication, a client (local host) must provide its signature with its private key so the server (remote host) can validate it with client's public key. In this scheme, an identity is now presented by signature rather than shared secret password. The servers only have the public key so they have nothing to leak.

Due to the fact that the client computer is also not trusted, a proper PKI implementation is difficult to accomplish, because the private key and PKI functions are vulnerable on the local host. Some implementations store the private key in a USB card, but they do not work for several reasons. First, if a USB card is only for storage, then PKI software must be able to access it; therefore, so can rootkits. Second, rootkits can attack those PKI functions by rewriting raw memory so that the compromised functions will not protect the private key. Third, rootkits can subvert the kernel I/O routines in order to make them disclose the private key.

More advanced implementations are using microprocessor-based smartcards. This kind of smartcard is an independent system that can store the keys and perform PKI operations on its own so viruses on the host computer cannot do anything. It does not solve the problem either, because an attacker can do phishing or forge an I/O message to trick the smartcard into doing something unexpected. For example, when a user purchases a $10 item on-line, malware can alter the message to trick the smartcard into signing a $100 transaction or signing to pay to a different payee. Also, whenever a smartcard is activated, rootkits can immediately ask it to perform several signatures for spurious purchases on the user's behalf. The transactions and signatures are totally genuine although the user did not authorize them.

In our solution, the SecCore is used to safe-keep the private keys (user's identity) aside from the local host computer. Host software only forwards the signature while the real signing is performed by the SecCore. This way the client and server do not keep any secrets and therefore it eliminates the risk of identity theft. Just like a user may have several different usernames and passwords for different identities, multiple

UNCORRECTED PROOF

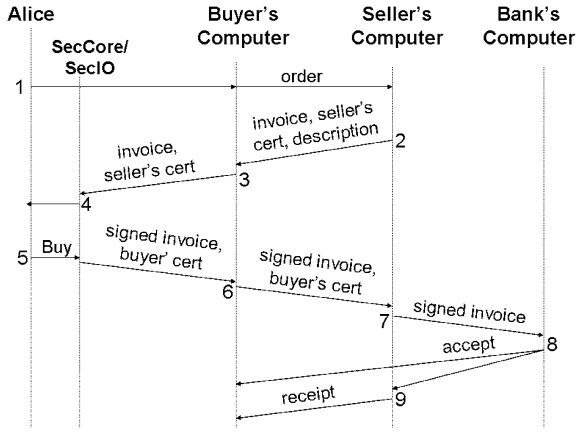

Fig. 3. PKI-based SecCore/SecIO for eBusiness.

keys for different user identities are possible. When the user needs a new identity, the SecCore generates a new PKI key-pair and the generated private key will never leave the SecCore, but the public key is freely distributed to anyone else. Of course, a user can generate as many PKI key-pairs as he wishes, but the created key does not have any meaning to others until it is mapped to a real digital identity certificate by an authority at a later time.

In addition to being a simple storage device, the SecCore and SecIO can perform a secure financial transaction while keeping any digital identity (the private key) secure. Our proposed protocol has fewer participants than a typical eCommerce model in order to simplify the discussion. Suppose Alice is an eShopper and she has the SecCore and SecIO installed in her computer. There are three host computers involved in a financial transaction; they are (1) the buyer's (or Alice's) computer, (2) the seller's (merchant's) computer and (3) the bank's (or broker's) computer. There are also a few assumptions. First, the bank is a trusted party in that its public key is known and pre-distributed to everyone. The bank had issued two different certificates – one to the buyer and another one to the merchant. Second, the buyer and seller are a customer of the same bank. Third, a new option, called *SecCore payment*, is supported by the participant's computers. Figure 3 shows a time line of the secure transaction protocol. Each operation is numbered for the discussion that follows.

1. Alice uses the web browser on her host computer to purchase some products from a seller, an on-line merchant. After filling the shopping cart, Alice checks the new option, "Using *SecCore payment* method".
2. The seller sends out an invoice, its certificate and description. The invoice, signed by the seller's private key, contains the merchant's name, amount, date and serial number. The certificate, issued by the bank, presents a valid identity of the seller. The description includes the shipping address, phone number and other trivial information.

UNCORRECTED PROOF

3. The browser displays all information regardless of whether the signature and certificate are valid or invalid. Since the browser supports the SecCore payment option, it forwards the invoice and certificate to its SecCore.
4. The SecCore validates the certificate using the already pre-distributed bank's public key. If valid, the SecCore extracts the seller's public key, verifies the signature and displays the invoice via SecIO. If any validation fails, the request will be rejected.
5. Alice reads the invoice shown by her SecIO and double checks the product description shown by the web browser. If Alice is ready to pay, she has the SecCore sign the invoice with her private key. Notice her command is entered from SecIO interface, rather than by any host software. After signing, the SecCore returns the signed invoice and Alice's certificate to the host browser.
6. The browser forwards everything to the seller.
7. Similarly, the seller validates the certificate and digital signature from the buyer. If they are valid, the seller sends the signed invoice to the bank for this agreed financial transaction.
8. The bank knows Alice's public key, so it can check the signature. The bank also checks Alice's account and its revocation list. An revocation can be any reason from bad credit to reported stolen SecCore. If all passed, then the bank admits this transaction between the buyer and seller and notifies both parties.
9. Once confirmed and accepted, the seller sends a receipt message and now it can proceed to the shipping and handling process.

The new protocol provides several protections to both the buyer and seller.

1. Alice's digital identity is well protected. There is no identity theft possible on her computer because the private key is sealed inside the SecCore.
2. Rootkits cannot make any legal signature, because the SecCore only accepts a signature request from its SecIO and Alice.
3. Rootkits cannot trick Alice into signing a different payee or amount, because the SecCore signs exactly what is displayed by the SecIO.
4. Alice is protected against any merchant look-alike phishing sites, because the phishers do not have a valid certificate to spoof the payee or amount.
5. Attackers cannot spoof any information on the network because of the signature; even the communication is protected.
6. Even the seller is protected against a bad charge or account number.

7. Social engineering

An open issue in using SecIO is human behavior. An important philosophy of developing security solutions is actually based on how to deal with people. Examples are a security policy, process or two-way authentication. Tricking users into releasing sensitive information or doing something against policy is a practice called

UNCORRECTED PROOF

social engineering. Popular social engineering attacks include the use of e-mail attachments and phishing. Similarly, tricking users to accept a malicious kernel patch is another social engineering attack. The best defense perhaps is back to education, which makes computer users more knowledgeable and powerful. For example, most users are aware of what viruses are, what damages they could cause, and they know not to open attachments or click links in suspicious emails.

All consumer software is intended to be user-friendly, efficient-to-use and easy-to-learn. Although in the consumer computing world, usability is paramount, that does not mean user interactions should be eliminated, especially since human supervision is a very powerful defense in making critical decisions. Computer users deserve better training about what to-do and not-to-do because they are the ones making a final decision. In fact, if people realize that education is the best way to protect them from identity theft or malware, they would probably like to get involved more, because no one wants to become a victim. For example, when a message like "invalid certificate" is displayed, and if users understand potential threats, they will not just blindly say "yes and continue anyway".

On the other side, software makers can make a better contribution. One way for them to do this is to provide a certified baseline integrity associated with their software. Today, more and more commercial software vendors are delivering software with a "code-signing package", which contains the certificate and digital signature for integrity. Another important factor is the operating systems. Since an OS runs in privilege mode, the kernel should be carefully designed and scrutinized. Today, the most popular way to install rootkits is through kernel vulnerability or a patch. Kernel patches are dangerous and should be avoided as much as possible. When such a patch is really needed, its vendor makes the news publicly available via other alternative media such as a newsletter so this information can better assist all end users. Perhaps it means extra costs, but considering most kernel patches are bug fixes, software vendors should share more responsibilities. There is no easy solution to social engineering problems, but with better education, training, security solutions and software vendors working together, it will make a difference.

8. System implementation

Our host computer is a common PC with a Pentium 4, 2 GHz 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 the Intel Pentium III CPU and VIA Apollo Pro266T chipset. The north bridge chip connects 2xAGP video (16 MB video RAM) and 256 MB system memory, while the south bridge chip supports USB ports, IDE controller,

UNCORRECTED PROOF

super I/O chip, 10/100 Ethernet chip and audio. The SlotServer is a single board computer and mainly used for the web server, firewall or distributed systems.

The configuration is close enough to our proposed platform. In a single box, it has two independent running systems, two sets of I/O devices (one physically attached to the SlotServer) and shared resources through a PCI bus. The host chipset maps the lower 64 kB of SlotServer's main memory, but on reverse side, the SlotServer only maps 4 kB of the host's memory due to its chipset. We do not have any tools to reset the shared PCI memory allowance in the chipset, however, because they are two independent symmetric systems, our workaround is to use the SlotServer as host computer and use the Pentium 4 machine as SecCore and SecIO. The SecCore runs the Linux 2.4.32 kernel and the host operating system runs the Linux 2.4.18 kernel, because one of the unmodified rootkits cannot run on a newer version of kernel. Both kernels are slightly modified. In SecCore, we added a set of functions to probe the 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 8.1, we briefly show the PCI architecture since the SecCore heavily depends on it. In Section 8.2 we present the data structure of the ELF binary which is needed by an integrity builder to find and compute an initial integrity. The builder can also be a system call so that any host software or end user can use it. In Sections 8.3 and 8.4, we describe the newly added routines in the host kernel – SecISR and checkers.

8.1. PCI system architecture

In general, most of the PCI devices can at least support 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 to attach 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 a 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 the PCI-shared memory region, the data is 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 chips are also known as a chipset. The chipset connecting

UNCORRECTED PROOF

PCI devices decides the memory range allowed to the CPU. Many of Intel's chipsets allow up to 64 kb + 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 manner is not a new technology. Several commercial products have merged the PCI-shared memory from each PCI SBC (Single Board Computer) into a tightly coupled cluster. In this cluster, the inter-node communications are going through PCI-shared memory across the PCI bus.

The SecCore needs a mechanism to communicate with the host CPU. Using a 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.

The bus mastering feature is optional. This technique enables the bus controller to communicate directly with other devices without the CPU's attention. It allows a device to be a master, so it can drive the data bus and directly read from/write to the memory bank and control signals. DMA is a simple form of bus mastering where the I/O device is set up by the CPU to access the memory block and then signal the 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 the host. However, this is not required, because the SecCore only needs to access a small memory block. Therefore, it does not cause a lot of performance impact.

8.2. 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, re-locatable 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 the 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 the Linux utility `md5sum.c` and specialized for ELF format. The basic logic of building the user-level software and kernel are almost identical. An application has exactly one baseline integrity, but a kernel is split into multiple baseline integrities which will be discussed later. The builder must call the SecCore to get the integrity data signed before storing them in a file. In this implementation, the builder uses host memory indicating a service request and also to get results from there instead of directly writing the command register and reading the status register in the SecCore.

UNCORRECTED PROOF

We briefly show how an executable is generated together with its internal format, because some of the information is related. In general, all programs are linked to a `/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 archive 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 time, the `sys_exec` is called by all `exec` wrappers. It uses a few service routines to allocate page frames, prepare data structures, get the `dentry`, file and `inode` object, copy arguments and environment variables, and scan the format to apply the corresponding method. In ELF format, the `load_binary` method is called to invoke the `do_mmap` function to create a new memory region that maps a text segment of the executable file. The initial linear address of the code by default starts from some default offset `0x08048000` in the application. The kernel's linker script `linux/arch/i386/vmlinux.lds` set the kernel code to start from `0xc0100000` by default. This data is essential for the checkers to locate the software image in virtual memory. Next, the virtual address needs one more translation to become a final physical address. Finally, `load_binary` sets the values such as the `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` will point to the new user stack, respectively.

Figure 4 shows a static memory instance of an executable and its in-memory image. The left side shows an ELF executable processed by the builder while generating an initial integrity. The right side shows a simplified running software image in memory that will be attested by the software checker. During the attestation, the checker verifies the signature, uses the data to locate its `.text` and compare its integrity.

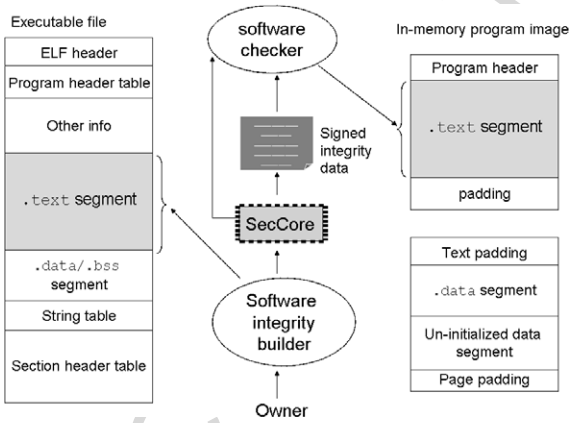

Fig. 4. Building and checking software integrity.

UNCORRECTED PROOF

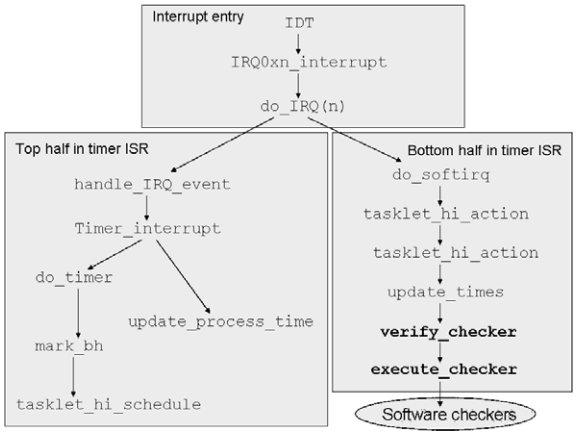

Fig. 5. Modified timer interrupt handler with SecISR embedded.

8.3. 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 Fig. 5. 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 the `IRQ0x00_interrupt`. Each interrupt vector has its own ISR, but they all share a common design and routines. Linux separated an ISR into top and bottom halves. The top half is for the most time critical portion and the bottom half is for non time critical portions that can be deferred. The kernel uses the generic facilities `do_IRQ`, `handle_IRQ_event`, `tasklet_hi_schedule` and `do_ softirq` to execute the top half and schedule the bottom half to run. In Fig. 5, the last function in timer ISR is our `wake_up_checker`, that starts a number of software checkers.

There are a 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 the timer runs, and the SecISR becomes the root-of-trust. However, since the SecISR can be tampered with by rootkits, not having SecCore is not secure. Second, the `wake_up_checker` needs to verify the checkers before invoking them. The integrity data are stored in a local file. It means that the `wake_up_checker` must perform additional file I/O operations. Third, the `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 the timer ISR and the checker can block their own events.

UNCORRECTED PROOF

8.4. Software checkers

Although it is possible to put all checking functions in one place, an all-in-one design would lack flexibility. We implemented a number of software checkers based on their purpose and specialty in the host kernel. The patched kernel can alternatively opt-in or opt-out using `make config`. Perhaps, there is has no clear answer on how many checkers are optimal, but we developed four checkers based on their characteristics and specialization. They are: (1) syscall 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 targets of today's rootkits. The syscall checker first attests the `sys_call_table` array where addresses of system calls are kept. Next, it validates the integrity of the `.text` for each system service routine. Although this data is part of the 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 special. 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 a few times. The `create_module` replaces all external and global symbols with corresponding logical addresses. The 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. This data is maintained in a linked list and used by the 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 that 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 do not verify approximately the first five pages in the `.text.init` section which 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 the checking operations are identical to other checkers.

9. 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

UNCORRECTED PROOF

against our own rootkit utility. Third, we evaluate the software updates. Forth, we measure the performance overhead on the host.

To evaluate whether our model can detect rootkit attacks, we used a few well known real-world kernel rootkits. These rootkits were obtained from the public domain and all were not modified. *Adore* and *Adore-ng* impact the kernel module and intercept the execution flow by altering the system call table and virtual file system. *SuckIT* presents a different attack, it is loaded through `/dev/kmem` by writing a special file to memory. We installed those rootkits one at a time, our checkers could successfully detect when Adore replaced the system call table, and when SuckIT rewrote the raw memory. The checker also discovered when Adore-ng removed itself from the module list, however, the checker did not recognize that Adore-NG illegally altered the VFS function pointers. We will discuss this limitation in Section 10.

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, 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 the 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 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 the SecCore side because its operations do not interfere with the host CPU. We measured that the performance impact on the 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 the case of large applications such as a database, there would be some extra cost for the kernel to page-fault all missing pages and bring them in due to user programs being on a page-on-demand basis. In a system where memory usage is high, additional required swap-in and swap-out could certainly

Table 1
Common rootkits and detection results

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

UNCORRECTED PROOF

worsen the system performance. However, this does not happen to kernel checking because the kernel itself will never be swapped out. Running the checking every five seconds was simply an appropriate choice because there are no known rootkits or viruses that repeat loading and unloading themselves within every five seconds.

10. 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 discussed in the 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 data is dynamic and is 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 the 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 the standard BFD (Binary File Descriptor) library as most GNU utilities do.

We also have two open issues to discuss.

1. The first concern is about social engineering. Regardless 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 certificates, and so on. In a software tamper resistant world, a solution must involve users, along with software and hardware vendors all together with security mechanisms.
2. The second one is the ease-of-use concern. This is especially true in the consumer domain. Usability is paramount. Our security mechanisms automatically run in the background and are unobtrusive. Also, the overhead cost is minimal. However, user interaction is really unavoidable in some cases such as recovery or updates. Computer users should be better educated about the importance and power of human supervision as a line of defense, even if it costs a little inconvenience.

UNCORRECTED PROOF

11. Conclusion

This paper presented a hardware enhanced approach to preserve integrity in operating systems and application software that protects computers from rootkits, viruses and other malware. Attesting software integrity is a difficult but critical task, and it is the cornerstone of 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 provides a flexible and robust security solution for software integrity assurance.

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

References

[1] F-Secure's Data Security Wrap-up for January to June 2006, http://www.f-secure.com/2006/1.

[2] Computer Virus Prevalence Survey, ICSA Labs 10th Annual, http://www.icsa.net/icsa/docs/html/library/whitepapers/VPS2004.pdf.

[3] McAfee Virtual Criminology Report, http://www.mcafee.com/us/local_content/misc/mcafee_na_virtual_ criminology_report.pdf.

[4] Sony, Rootkits, and Digital Rights Management Gone Too Far, http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rghts.htm.

[5] R. Naraine, Symantec caught in Norton 'Rootkit' Flap, Technical report, 2006.

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

[7] LaGrande Technology for Safer Computing, http://www.intel.com/technology/security.

[8] Microsoft Next-Generation Secure Computing Base – Technical FAQ, http://www.microsoft.com/technet/archive/security/news/ngscb.mspx.

[9] Trusted Computing Group, http://www.trustedcomputinggroup.org/.

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

[11] CERT Research 2005 Annual Report, CERT Research and the Rapidly Evolving Threat, http://www.cert.org/archive/pdf/cert_rsch_annual_rpt_2005.pdf.

[12] D. Sancho, and J. Jamz, 2006 Annual Threat Roundup and 2007 Forecast – A Special Report by Trend Micro, 2006, http://www.trendmicro.com.au/global/products/collaterals/white_ papers/2006AnnualThreatRoundup.pdf.

UNCORRECTED PROOF

[13] McAfee Inc., Rootkits, Part 1 of 3: The Growing Threat, 2006, http://www.mcafee.com/us/threat_center/white_paper.html.

[14] Symantec Corp., Internet Security Threat Report 2006, 2006.

[15] IBM report: Surge in criminal-driven cyber attacks anticipated in 2006, http://www-03.ibm.com/industries/financialservices/doc/content/news/pressrelease/1500860103.html.

[16] Rootkits, Part 1 of 3: The Growing Threat, www.mcafee.com/us/local_content/white_papers/threat_center/wp_ akapoor_rootkits1_en.pdf.

[17] F. Cohen, Operating system evolution through program evolution, *Computers and Security* (1993).

[18] D.M. Chess and S.R. Whit, An undetectable computer virus, in: *Virus Bulletin Conference*, Sept. 2000.

[19] K. Thompson, Reflections on Trusting Trust, *Communications ACM* **27** (1984).

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

[21] K. Wika and J. Knight, A safety kernel architecture, Technical report, 1994.

[22] R. Farrow, The kernelize secure operating system (KSOS), in: *USENIX and SAFE Magazine, Inside: Security*, Dec. 2002.

[23] A. Tanennaum, J. Herder and H. Herbert Bos, Can we make operating systems reliable and secure?, *IEEE Computer Society* (2006).

[24] V. Basili and B. Perricone, Software errors and complexity: An empirical investigation, *Commun. ACM* (1984).

[25] T. Ostrand and E. Weyuker, The distribution of faults in a large industrial software system, in: *Proc. Intl. Symp. Software Testing and Analysis*, ACM Press, 2002.

[26] A. Chou et al., An empirical study of operating system errors, in: *Proc. 18th ACM Symp. Operating System Principles*, 2001.

[27] M. Swift, B. Bershad and H. Levy, Improving the reliability of commodity operating systems, *ACM Trans. Computer Systems* (2005).

[28] W. Arbaugh, D. Farber and J. Smith, Secure and reliable bootstrap architecture, in: *Proc. IEEE Symposium on Security and Privacy*, May 1997.

[29] Y. Wang, C. Verbowski, H. Wang and J. Lorch, SubVirt: Implementing malware with virtual machines, in: *Proc. 2006 IEEE Symposium on Security and Privacy*, 2006.

[30] Against TCG, http://againsttcpa.com/.

[31] Press release – Microsoft Palladium: A Business Overview, http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp.

[32] R. Goldberg, Survey of virtualization machine research, *IEEE Computer* (1974).

[33] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum and D. Boneh, Terra: A virtual machine-based platform for trusted computing, in: *Proc of 19th Symposium on Operating System Principles (SOSP)*, 2003.

[34] G. Dunlap, S. King, S. Cinar, M. Basrai and P. Chen, Revirt: Enabling intrusion analysis through virtual-machine logging and replay, in: *Proc of 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[35] T. Garfinkel and M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: *Proc. Network and Distributed Systems Security Symposium*, 2003.

[36] Y. Wang and P. Dasgupta, VMM based security assurance, PhD thesis, Arizona State University, 2007.

UNCORRECTED PROOF