Hardware-Assisted Application Integrity Monitor

Jiang Wang, Kun Sun, Angelos Stavrou
Center for Secure Information Systems,
George Mason University Fairfax, VA 22030
{jwanga, ksun3, astavrou}@gmu.edu

Abstract

Existing hardware-assisted methods monitor the integrity of hypervisors and operating systems, which are critical to system integrity. This protection is possible because of "non-volatile" data structures present in the machine's physical memory. In contrast, applications offer a more challenging protection target because they are dynamically allocated. Therefore, robust defenses against application tampering is still a difficult tasks that has remained an open problem.

We propose "AppCheck," a hardware-assisted framework the protects the integrity of applications and server processes. We achieve that by leveraging semantic information extracted from the source code and input from a human developer. Unlike pure software defenses, AppCheck employs existing x86 features, namely System Management Mode, to acquire the necessary memory contents. If any of the these critical components become altered during runtime, AppCheck signals an alarm to a remote server notifying the operators of a potential security breach or software corruption.

1 Introduction

The code size and functional complexity of modern desktop applications and back-end servers keep increasing leading to an increased attack surface and risk for intrusion. For example, Web browsers including the popular Firefox and Internet Explorer are known to have a large set of software components and plugin modules that come build-in. Moreover, office applications such as OpenOffice and Microsoft Office have evolved over time to support an ever increasing set of functions. All of these large and highly sophisticated applications provide many capabilities and features which make them among the most commonly used today. At the same time, it also makes them targets for adversaries, who employ zero-day vulnerabilities aiming to take over applications with the goal of compromising the entire machine.

To combat malware, researchers have developed and deployed a wide-range of protection mechanisms. Some of these defenses are malware-specific, such as signature-based intrusion detection or anti-virus systems. Others are application-specific: for example, Abadi et al. have attempted to enforce the control-flow integrity for the applications [5]. Recently, there have been efforts to leverage some of the isolation properties provided by virtualization teachnologies to guard the integrity of systems by separating applications into different containers [31].

Furthermore, researchers have proposed hardware-assisted methods to validate the integrity of the operating system kernels [32, 6, 26]. In their implementation prototypes, they employ either specialized hardware (in the form of a PCI card), a common network card, or the System Management Mode (SMM) of the x86 CPU to monitor the integrity of the running software stack. Here, we try to extend this hardware-assisted kernel integrity protection to safeguard instances of user-level applications as well. Unlike the software stack of operating system kernels, which is mostly static and immutable when being mapped into the physical memory, the user-level applications are dynamically loaded by the operating system into memory locations that have to be semantically recognized. Therefore, their locations within the physical memory may be different for each instance. Our approach is protects data in physical memory rather than virtual memory because hardware, such as a PCI card, can only access the physical memory without requiring complex page translation and calculations. However, for open source operating systems, we could potentially support protection of data stored in virtual memory.

In this paper, we develop an application integrity monitor using a hardware-assisted method. More specifically, we monitor the code integrity of any applications that interest the user. To achieve this, we use the SMM to scan the physical memory and obtain the code section of a designated user-level process. We then use a commercial network card to send the contents to a remote server where another program can verify the code integrity of the application.

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.299

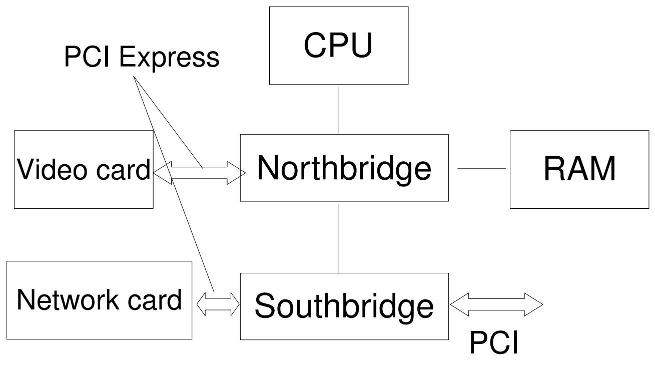

Figure 1: Typical hardware layout of a computer

2 Background

In this section, we provide a detailed background of the typical architecture of an x86 PC, System Management Mode, and the basic knowledge of the BIOS.

2.1 Typical Computer Architecture

In Figure 1 we depict the internal architecture of a computer with one processor. The top is the Central Processing Unit. Northbridge connects the CPU with the RAM and the video card. Southbridge is connected with Northbridge, the PCI bus, and other devices. AppCheck uses a dedicated PCI Ethernet card, which is connected to Southbridge to read the physical memory. Note that both Northbridge and Southbridge provide PCI device interfaces so that they can be accessed and configured through PCI configuration methods. Recently, a PCI Express [25] bus standard was developed to replace the conventional PCI, PCI-X and AGP bus. PCI Express could be connected with Northbridge and Southbridge. PCI Express also introduces a much higher transfer rate than PCI. The highest transfer rate for PCI Express 2.0 is 5.0GT/s, while conventional PCI is normally only 133MB/s (32bit at 33MHz). The high speed of PCI Express provides an opportunity for AppCheck to finish scanning the physical memory quickly with relatively low overhead.

2.2 System Management Mode

Intel IA-32 CPU provides three operation modes: real-address mode, protected mode, and system management mode [13]. Intel x86-64 CPU supports all three of these modes and adds one more: IA-32e. CPU enters real-address mode following system power-up or reset. The modern operating system then switches to protected mode. System management mode provides an operating system or executives with a transparent mechanism for implementing platform-specific functions, such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or when an SMI is received from the advanced programmable interrupt controller (APIC). The CPU can enter in SMM either from the real-address mode or from protected mode, as shown in Figure 2. For current operating systems, such as Windows and Linux, the CPU runs in protected mode most of the time. Switching to SMM rarely happens.

In SMM, the processor switches to a separate address space while saving the basic context of the currently-running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is restored to the state before the system management interrupt is triggered. SMM was introduced with the Intel386 SL and Intel486 SL processors and became a standard IA-32 feature with the Pentium processor family [13]. The separate address space used by SMM is called the system management RAM (SMRAM). The memory addressing model is similar to that in the real-address mode. SMM is designed to be transparent to the operating system, and the SMRAM can be made inaccessible from other operating modes. We use this feature of SMM to run the register integrity checking code and to reliably drive the NIC.

2.3 BIOS, UEFI and Coreboot

The system BIOS is an omnipresent and indispensable component for all modern personal computers and

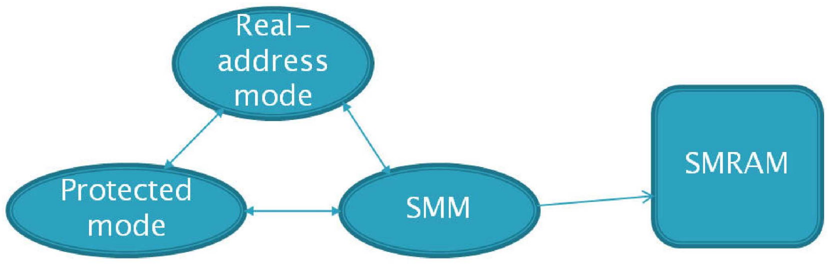

Figure 2: x86 CPU mode transition

servers. The main function of the BIOS is to initialize the hardware, including the processor, main memory, northbridge, southbridge, hard disk, and other necessary IO devices, such as the keyboard. The entire process is referred to as the Power-On Self-Test (POST). The BIOS initializes enough hardware so that it can transfer control to an operating system or bootloader. SMM code is also set up by the BIOS. BIOS code is normally stored on a non-volatile ROM chip built into the system on the motherboard.

BIOS manufacturers have traditionally provided the BIOS implementations in assembly language because it has to be able to fit in relatively small memory space and operate as fast as possible in real-address mode. In recent years, a new generation of BIOS, referred to as the Unified Extensible Firmware Interface (UEFI) [4], has become increasingly popular in the market. UEFI is a specification that defines a new software interface between OS and firmware. One purpose of UEFI is to ease development by switching to protected mode in a very early stage and writing most of the code in C language. A portion of the Intel UEFI framework (the Tiano Core) is open source; however, the main function of the UEFI (i.e., to initialize the hardware) is still closed source.

Coreboot [2] (formerly known as LinuxBIOS) is an open-source project aimed at replacing the proprietary BIOS (firmware) in most of today's computers. It performs a small amount of hardware initialization and then executes a so-called payload. In some sense, coreboot is similar to the UEFI-based BIOS. Coreboot also switches to the protected mode in a very early stage and is written mostly in C language. Our prototype implementation is based on coreboot V4.

3 Threat model

3.1 Adversarial capabilities

We assume the adversary is able to exploit the remotely or locally vulnerabilities present in any software running in the machine after boot-up. This includes all the applications, device drivers, operating system kernels, and hypervisor code, if present. Moreover, the adversary is potentially capable of modifying the application code or data using known or zero-day attacks leading to compromises of the running software stack of the machine.

3.2 General Assumptions

The attacker cannot tamper with or replace the installed PCI NIC with a malicious NIC using the same driver interface. Furthermore, if the SMM code is integrated with BIOS, we assume that the SMRAM is properly set up by BIOS upon boot time. If the SMM code is not included in the BIOS, it must be reliably uploaded to the SMRAM during boot-up. This can be done by using either the trusted boot or the management interface to bootstrap the computer. In this case, to initialize the SMM code, a trusted bootstrap mechanism must be employed. The SMRAM is locked once it is properly set up. Once it is locked, we assume that it cannot be subverted by the attacker, and this assumption is supported by current hardware. Attacks that attempt to modify the SMM code [35, 15, 16] are beyond the scope of this paper.

3.3 In-scope Attacks

AppCheck aims to detect in-memory, user-level code modification attacks, even when the whole software stack (including the operating system kernels) is compromised. If the operating system kernel is not compromised, then the kernel can check the code integrity of the applications. However, the kernel itself may be compromised. The kernel is also a piece of software and normally has a large size. Therefore, it contains some vulnerabilities that can be exploited by attackers. Once the kernel is compromised, there is nothing left to be trusted aside from the hardware. Existing hardware-based protection mechanisms target only the kernels; how to extend them to the user-level processes has remained unclear. We try to bridge this gap through AppCheck.

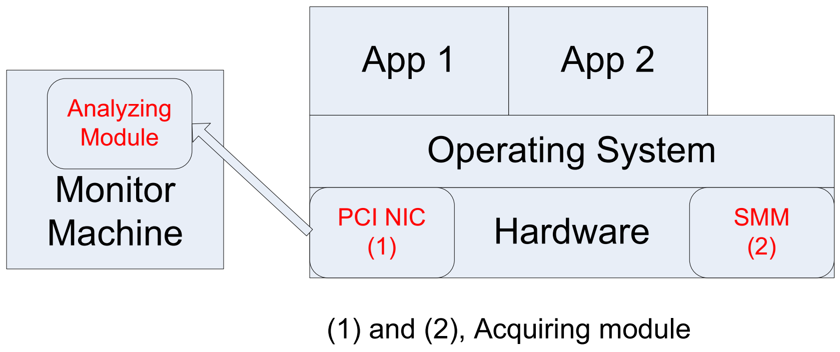

Figure 3: The architecture of the AppCheck

3.4 Limitations

Currently, our solution cannot protect against attacks that modify dynamically allocated data. With altering dynamically allocated data we refer to two classes of threats: (1) modification of the dynamically-generated function pointers inside an application, if present or supported and (2) return-oriented attacks. In return-oriented attacks, the control flow of the benign application is hijacked and redirected to a memory location controlled by the attacker. Of course, there are other proposed techniques exist that thwart such attacks individually: the non-executable bit in new CPUs and Address Space Layout Randomization, to name a few. AppCheck can offer lower-level integrity checking because, by virtue of running directly in hardware using only BIOS support, it cannot be subverted. Therefore, we can leverage and integrate any software defenses to provide Multi-Layer Security (MLS) protection. In this paper, however, we did not leverage any of these additional protection schemes but this is not a limitation of the proposed framework.

4 Overall System Architecture

The architecture of AppCheck is shown in Figure 3. The right side depicts the target machine that is protected. The left side depicts the remote machine that monitors the application integrity of the target machine. AppCheck is comprised of two components: the acquiring module and the analysis module. The acquiring module includes the PCI NIC and the SMM of the CPU. The analysis module resides on the remote machine and connects to the acquiring module via a network cable. The logic steps of AppCheck are shown in Figure 4.

4.1 State Acquisition

The acquiring module has two main functions: to locate the contents in the physical memory that should be monitored, and to acquire those contents. Our acquiring module includes the SMM and a PCI network card. The SMM can be triggered periodically.

4.1.1 Locating memory

Locating the contents in the physical memory is a challenge since AppCheck is a hardware-based mechanism and many operating systems and user process-level contexts may not be available for the hardware. For example, a user-level application on Linux can use "ps" command to find all of the running processes. A Linux kernel module can use the "proc" file system to discover the same information. On Microsoft Windows, the user-level application can use the $CreateToolhelp32Snapshot$ and $Process32First$ functions to obtain the processes information. However, none of these are available for the hardware (SMM or NIC). This is known as the "semantic gap" problem [12].

To solve this issue, the hardware must obtain the semantic knowledge of the running operating system and the applications. There are two ways to obtain this knowledge. The first is to rely on some expert's knowledge about the operating system and to allow the expert to tell the system where the location (i.e., physical memory address) is for those contents. The second method is to automatically derive that knowledge through either the source code [7] or through some other debug information or running traces. For those operating systems that have the source code available, a special compiler can be used to recompile the source code and obtain the necessary information. CIL [24] is such a tool for C languages. For those operating systems without the source code, such as Microsoft Windows, the debug information (e.g., the symbol files in Windows) may be used for the same purpose.

4.1.2 Translating the physical memory

The results of the previous step are normally in virtual address format. For example, if the previous step tells us that the *current* variable on Linux is at 0xc3a94000, then we would need to ascertain the physical address that corresponds to that virtual address since the hardware (PCI NIC) only knows the physical one.

For this purpose, we use the SMM to obtain the control register 3 (CR3) on the CPU. CR3 contains the base physical address of the page tables. From CR3, we can discover and walk through the page tables and calculate the physical address if given a virtual one.

4.1.3 Acquiring the physical memory

Another function of the acquiring module is to safely and reliably acquire the contents.

In general, there are two ways to acquire the physical memory: software and hardware methods. The software method uses the interface provided by the operating system to access the physical memory, such as `/dev/kmem` on Linux [10] or \Device \PhysicalMemory on Windows [30]. This method relies on the integrity of the underlying operating system. If the operating system is compromised, then the malware may provide a false view of the physical memory. Moreover, these interfaces for accessing memory can be disabled in future versions of the operating systems. In contrast, the hardware method uses a PCI device [11, 26] or other kinds of hardware [9]. This method is more reliable since it is less dependent on the integrity of the operating system.

We choose the hardware method to read physical memory. There are also multiple options for choosing the hardware components, such as a PCI device, a FireWire bus device, or a customized chipset. We use a PCI device because it is the most commonly used hardware. Moreover, existing commercial Ethernet cards need drivers to function. These drivers normally run in the operating system, which are vulnerable to attacks and may be compromised in our threat model. To avoid this problem, AppCheck puts these drivers into the SMM code. Since the SMRAM memory will be locked after booting, it will not be modified by the attacker. In addition, to prevent the attacker from using a malicious NIC driver in the OS to spoof the SMM driver, we use a secret key. The key is obtained from the remote machine when the target machine is booting up and then stored in the SMRAM. The key is then used as a random seed to selectively hash a small portion of the data in order to avoid data replay attacks.

When using the hardware-based method to protect the operating system kernel, the contents of the kernel code are always in the physical memory. For user-level processes, some of the pages may be paged out to the hard disk. In that case, the PCI network card cannot obtain the contents on the hard disk. To overcome this issue, we check the "present" bit in the page tables and only monitor those pages that reside in the physical memory.

4.2 Analysis Module

The analysis module resides on the remote server. It requires two steps: (1) to receive and filter the network packets from the target machine, and (2) to process those packets. For the first step, the module could use the raw sockets provided by Linux, the libcap library, or the tcpdump program. After the module obtains the network packets, it needs to filter out the packets not intended for the analysis module. Libcap and tcpdump already provide such filters.

For the second step, the module has two options. First, if the module is simply monitoring the static code part of an application, it can perform a bit-by-bit comparison for each application memory snapshot. If the two snapshots are different, then this indicates a potential attack. The limitation of this method is that it cannot monitor other dynamic contents. Second, to monitor the dynamic contents, the analysis module could use some tools mentioned in Section 4.1.1. For example, the analysis module can perform tasks similar to the one in [7].

5 Case Study: Protecting a Browser

In this section, we describe a case study that uses AppCheck to protect a Firefox browser process. We use Firefox as an example because it has a sizable code base and is frequently-attacked by malware and remote exploits. Of course, our method is more general and can be applied to detect tampering of data on other applications and server processes. For our experiments, we used CentOS [1] 5.4 (x86 32bit) as an operating system running Firefox version 3.0.5.

5.1 SMI Triggering

An SMI can be triggered in many ways. For example, according to the manual of Intel ICH [3], writing to the port 0xB2 will trigger an SMI. A hardware timer can trigger an SMI too. In addition, the PCI network card can trigger an SMI when it receives a packet. We used the latter method, which is also used by HyperCheck [32].

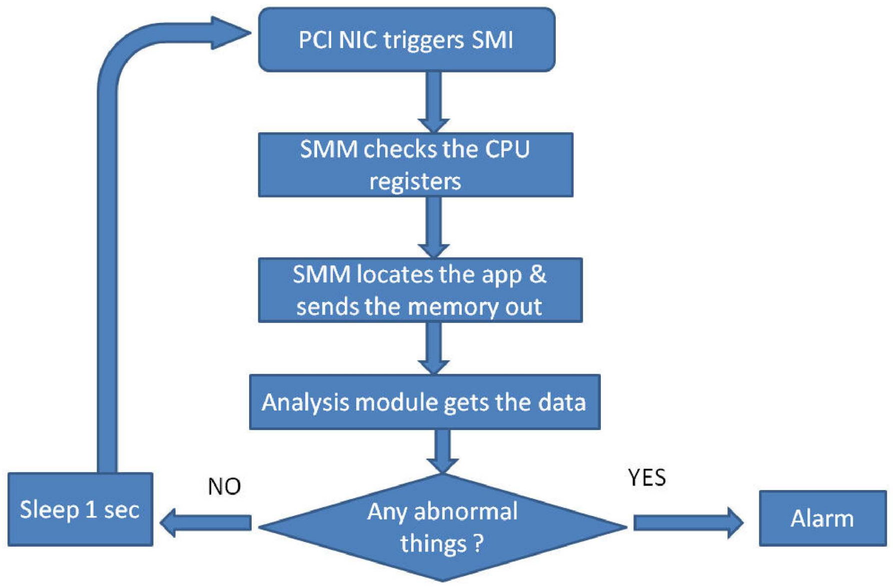
PCI NIC triggers SMI
SMM checks the CPU registers
SMM locates the app & sends the memory out
Analysis module gets the data
Any abnormal things ?
NO
Sleep 1 sec
YES
Alarm

Figure 4: Logic steps of the AppCheck

5.2 Locate Firefox Instance

When the SMM is triggered, the current running process may not be the Firefox process. One naive method is to simply exit from the SMM and hope that the subsequent running process would be Firefox. However, there is no guarantee of this and, in a worst-case scenario, the SMM may never catch the Firefox process.

To solve this problem, we try to find the Firefox process information even if it is not the current process. This is feasible since all of the processes are linked in a doubly-linked list on Linux (Windows has similar link lists). More specifically, Linux uses the *current* variable to denote the current process. It is a $thread\_info$ structure pointer, and the first member points to the *task_struct*, which denotes each process on Linux. The *current* variable is stored at the bottom of the kernel stack. To reach it, we first obtain the *esp* value of the running process from the SMRAM saved area. Supposing that this value is 0xc1004566, then the *current* would be 0xc1004000, which is obtained by setting the lower 13 bits of the *esp* to zero.

We have now obtained the *current* and know that its first member is the *task_struct*. The next step is to check whether the current process is Firefox or not. If not, then we obtain the name of the current process by using the *comm* member of the *task_struct*. The next question is how to find the offset of *comm* member. There are two ways to solve this problem. One is to use the CIL [24] to process the kernel source file and automatically calculate the offset. The other way is to write a kernel module and print out the offset during the runtime. We use the second method and find that the offset was 0x1AC.

With the knowledge of the offset for *comm*, we can ascertain its value and verify whether it contains the Firefox string or not. Note there may be multiple Firefox processes. If there is no such string, then we move on to the next process by following the *tasks* member of the *task_struct* and continuing the check.

5.3 Locate Process Code

Once we have located the *task_struct* of the Firefox process, the next step is to locate the virtual address of the Firefox code section. For this purpose, we use the memory descriptor *mm*, which is a field of the process descriptor (i.e. *task_struct*). The type of *mm* is *mm_struct*, and it contains two files named *start_code* and *end_code*, which have a type of *unsignedlong*. These two fields denote the virtual address of the beginning and ending of the code part of the process. We can use them to obtain the actual code used by the Firefox process. The offsets for these fields within the structure can be obtained by using similar techniques to those mentioned above.

5.4 Acquire Process State

Virtual addresses are used by the operating system to manage the process. However, since AppCheck uses the PCI network card to acquire memory, it understands only the physical memory. Therefore, AppCheck must translate those virtual addresses to physical ones. To achieve this, AppCheck needs to know the CR3 (page directory base register). When the CR3 is known, the translating process basically walks through the page tables the same way as the hardware and discovers the physical address corresponding to the virtual one. To obtain this register, we used the SMM. Before switching to the SMM, the CPU hardware automatically saves the current CPU registers to the SMARM. Then, the SMM code can obtain the CR3 register from the SMRAM and discover the physical address of the Firefox code section. Different processes have different CR3 values, which are stored in *task_struct* of each process. Next, the SMM code drives the network card to access the Firefox code section through DMA and then sends it out to the remote server.

5.5 Snapshot Analysis of Memory

The analyzing module running on the remote server receives the packet from the network card and then can analyze the code section of Firefox. Since the code section is static, the analyzing module simply performs a bit-by-bit comparison between the current code section and the its previous clean snapshot. If they are different, a potential attack is identified. We do not monitor the dynamic data (e.g., the data allocated in the stack or the heap); however, we could use some tools, such as CIL, to analyze the invariance of this data.

6 Related work

Using hardware-assisted techniques to protect software from integrity attacks is not a new concept: researchers have used a special-purpose PCI device in the past to acquire physical memory, either for rootkit detection [26, 7] or for forensic purposes [11]. The systems that align most closely with our research are Copilot [26] and HyperCheck [32]. Copilot employs a special PCI device to poll the physical memory of the host and periodically send it to an admin station. HyperCheck does not require specialized hardware, but an out-of-the-box network card. HyperCheck also uses the SMM and a network card to offer a complete view of the machine state, including CPU registers and memory. However, HyperCheck can only protect the integrity of hypervisors or operating systems, whereas AppCheck can protect the intergrity of applications as well.

Another closely related work is HyperGuard [28]. Rutkowska *et al.* suggested using the SMM of the x86 CPU to monitor the integrity of the hypervisors. AppCheck has a different goal in mind: to monitor application integrity. Since the application may be created dynamically and allocated memory space, it is more difficult to monitor. We use the semantic information to find the location of the target application address and then acquire its memory. In addition, the use of a network card allows us to outsource the analysis of the state snapshot. This results in a drastic improvement in system performance, reducing system busy time from seconds to milliseconds. AppCheck also differs from Hyperguard in its ability to use the monitoring machine to detect DoS attacks against the SMM code.

DeepWatch [9] also offers detection of hypervisor rootkits, which it refers to as virtualization malware, by using the embedded micro-controller(s) in the chipset. DeepWatch is signature-based and relies on hardware-assisted virtualization technologies such as Intel VT-d [19] to detect rootkits. In contrast, AppCheck performs anomaly detection and aims to protect the applications.

Flicker [22] uses a TPM-based method to provide a minimum Trusted Code Base (TCB), which can be used to detect modification to the kernels. Flicker requires advanced hardware features. such as Dynamic Root of Trust Measurement (DRTM) and late launch. In contrast, AppCheck uses the static Platform Configuration Registers (PCRs) to secure the booting process. In addition, by sending out the data, AppCheck has a lower overhead on the target machine than Flicker. To reduce Flicker's overhead, TrustVisor [23] has a small footprint hypervisor to perform some cryptography operations. However, all of the legacy applications should be ported to work on TrustVisor.

There is also a plethora of research aimed at protecting the Linux kernel [7, 21, 18, 33, 20, 29, 27]. Baliga [7] *et al.* use a PCI device to acquire the memory and automatically derive the kernel invariance. We currently discover the kernel invariance manually, but we could employ their techniques without modifications. Litty [21] *et al.* developed a technique to discover the address of key data structures that are instantiated during run-time by relying on processor hardware and executable file specifications. However, they also rely on the integrity of the underlying hypervisors. AppCheck first obtains the virtual addresses by analyzing the source code and then calculates the physical addresses through CPU registers. Therefore, AppCheck can obtain the correct view of the system memory, even if the underlying OS or hypervisor is compromised and the page tables are altered.

Some recent work has gone towards using the SMM to generate efficient rootkits [34, 8, 17, 14]. These rootkits can be used either to obtain root privilege or as keystroke loggers. We use SMM to offer integrity protection by monitoring the state of the applications.

7 Conclusion

In this paper, we present a hardware-assisted method to monitor the application integrity. By employing a remotely triggered capability to switch to CPU SMM and a network card, our method guarantees a correct snapshot view of the state of the instance for applications. The application state can be extracted and inspected on a remote server even if portion or the entire software stack that runs on the machine becomes compromised or unavailable due to failures. Moreover, AppCheck can monitor dynamically spawned processes and applications by interpreting and relating the semantic information obtained from the source code with the physical footprint of the application instance in physical memory. This approach extends previous research on tamper detection mechanisms for protecting static data structures in kernels to user-land processes.

Acknowledgments

We would like to thank Fengwei Zhang at Center for Secure Information Systems at George Mason University. He helped us to setup the experiment environment.

This work was supported in part by the United States Air Force Research Laboratory (AFRL) through contract FA8650-10-C-7024, NSF grant CNS-TC 0915291, and a research gift from Google Inc. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government and sponsoring agencies.

References

[1] CentOS, http://www.centos.org/.

[2] Coreboot, http://coreboot.org/.

[3] Intel corp. intel i/o controller hub 9 (ich9) family datasheet (2008).

[4] Unified Extensible Firmware Interface, http://www.uefi.org/home/.

[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[6] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, 2010.

[7] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[8] BSDaemon, coideloko, and D0nAnd0n. System Management Mode Hack: Using SMM for "Other Purposes". *Phrack Magazine*, 2008.

[9] Y. Bulygin and D. Samyde. Chipset based approach to detect virtualization malware a.k.a. DeepWatch. *Blackhat USA*, 2008.

[10] M. Burdach. Digital forensics of the physical memory. *Warsaw University*, 2005.

[11] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50 – 60, 2004.

[12] P. Chen and B. Noble. When virtual is better than real. In *In 8th Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[13] I. Corporation. Intel® 64 and ia-32 architectures software developers manual volume 1.

[14] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. In *Proceedings of the 7th CanSecWest conference*. Citeseer, 2001.

[15] L. Duflot, D. Etiemble, and O. Grumelard. Security issues related to pentium system management mode. In *Cansecwest security conference Core06*, 2006.

[16] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM Reloaded. *CanSecWest, Vancouver, Canada*, 2009.

[17] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 11. ACM, 2008.

[18] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[19] R. Hiremane. Intel® Virtualization Technology for Directed I/O (Intel® VT-d). *Technology© Intel Magazine*, 4(10), 2007.

[20] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, page 138. ACM, 2007.

[21] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[22] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

[23] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[24] G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[25] PCI-SIG. PCI Express 2.0 Frequently Asked Questions.

[26] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[27] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2008.

[28] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[29] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, page 350. ACM, 2007.

[30] T. Vidas. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice*, 1(4):315–323, 2006.

[31] J. Wang, Y. Huang, and A. Ghosh. SafeFox: A Safe Lightweight Virtual Browsing Environment. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10. IEEE, 2010.

[32] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[33] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554. ACM, 2009.

[34] F. Wecherowski and core collapse. A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. *Phrack Magazine*, 2009.

[35] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel® CPU Cache Poisoning, 2009.