This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



# Detecting and Preventing Kernel Rootkit Attacks with Bus Snooping

Hyungon Moon, Hojoon Lee, Ingoo Heo, Kihwan Kim,
Yunheung Paek, *Member, IEEE,* and Brent Byunghoon Kang, *Member, IEEE,*

**Abstract**—To protect the integrity of operating system kernels, we present *Vigilare system*, a kernel integrity monitor that is architected to snoop the bus traffic of the host system from a separate independent hardware. This *snoop-based monitoring* enabled by the Vigilare system, overcomes the limitations of the *snapshot-based monitoring* employed in previous kernel integrity monitoring solutions. Being based on inspecting snapshots collected over a certain interval, the previous hardware-based monitoring solutions cannot detect *transient attacks* that can occur in between snapshots, and cannot protect the kernel against permanent damage. We implemented three prototypes of the Vigilare system by adding *Snooper* hardware connections module to the host system for bus snooping, and a snapshot-based monitor to be comared with, in order to evaluate the benefit of snoop-based monitoring. The prototypes of Vigilare system detected all the transient attacks and the second one protected the kernel with negligible performance degradation while the snapshot-based monitor could not detect all the attacks and induced considerable performance degradation as much as 10% in our tuned STREAM benchmark test.

**Index Terms**—Transient Attack, Hardware-based Integrity monitor, Kernel Integrity Monitor

## 1 Introduction

To protect the integrity of operating system kernel against rootkits, many researchers strive to make their monitors independently isolated from the monitored host system. Recent efforts on this kernel integrity monitoring can be categorized into two groups: hardware based approaches [1], [2] and hypervisor based approaches [3], [4]. While approaches based on hypervisors have gained popularity recently, hypervisors themselves are exposed to numerous software vulnerabilities [5], [6], [7], [8] as hypervisors are becoming more and more complex. Several approaches [9], [10] noted that inserting an additional software layer to protect the integrity of hypervisors may not be sufficient. The additional layer will introduce new sets of vulnerabilities in a similar fashion of the hypervisors; inserting another padding with a software layer for security may enhance security temporarily, but it does not provide a fundamental solution. As a solution for this, they introduced hardware-supported schemes to monitor the integrity of hypervisors.

Most of the existing hardware-based solutions to kernel integrity monitoring make use of snapshot analysis schemes; they are usually assisted by some types of hardware components that enable saving of the memory contents into a snapshot, and then perform an analysis to find the traces of a rootkit attack.


- *Y. Paek, B. Kang are corresponding authors. (ypaek@snu.ac.kr, brentkang@kaist.ac.kr) .*
- *H. Lee is the co-first author.*
- *H. Moon, I. Heo, and Y. Paek are with Seoul National University.*
- *H. Lee, K. Kim and B. Kang are with Korea Advanced Institute of Technology.*


Copilot [1], HyperSentry [9], and HyperCheck [10], [11] are exemplary approaches on snapshot-based kernel integrity monitoring. A custom Peripheral Component Interconnect (PCI) card to create snapshots of the memory via Direct Memory Access (DMA) in Copilot, and the System Management Mode (SMM) [12] are utilized to implement the snapshot-based kernel integrity monitors in HyperCheck and HyperSentry.

Snapshot-based monitoring schemes in general have inherent weaknesses mainly because they only inspect the snapshots collected over a certain interval, missing the evanescent changes in between the intervals. Attackers can exploit this critical limitation of snapshot-based kernel integrity monitoring. If attackers know the existence and the interval of snapshot-taking, they could devise a stealthy malware that subverts the kernel only in between the snapshots and restores all modification back to normal by the time of the next snapshot interval. This is called as *scrubbing attack*, and HyperSentry [9] addressed this by making it impossible for the attackers to predict when the snapshots will be taken. However, attackers can still create a transient attack that leaves its traces as minimal as possible without knowing the exact time that snapshot is taken. If the traces are left in the memory for a short time, there is a chance that it can avoid being captured in snapshots and not detected, as the authors of HyperSentry were aware of. The term transient attack refers to attacks which do not leave persistent traces in memory contents, but it still achieves its goal by using only momentary and transitory manipulations. In addition, such snapshot-based schemes cannot prevent the modification of

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



kernel code or data in the main memory. For this reason, they cannot find the original memory contents any place in the system, which is required to resotre the malicious memory contents to the genuine ones. Once the monitor detects any malicious modification, the only prescription for the compromised kernel is rebooting the system and exploiting the secure boot process [13], [14].

In this paper, we propose Vigilare, a snoop-based integrity monitoring scheme that overcomes the limitations of existing hardware-based kernel integrity monitoring solutions. The Vigilare integrity monitoring system takes a fundamentally different approach; it monitors the operation of the host system by "snooping" the bus traffic of the host system from a separate independent system located outside the host system. This provides the Vigilare system with the capability to observe all host system activities such as writing to the main memory, before the activities take effect, and yet being completely independent from any potential compromise or attacks in the host system. This snoop-based architecture enables security monitoring of virtually all system activities, as all processor instructions and data transfers among I/O devices, memory, and processor must go through the system's bus. By monitoring this critical path, Vigilare system acquires the capability to observe all activities to locate malicious system transactions, and prevent these transactions from being completed.

This Vigilare system is composed of the following components. *Snooper* on the Vigilare system is connected to the system bus of the main system, collects the contents of real-time bus traffic, and delivers the accrued bus traffic to *Verifier*. The main functionality of Verifier is to examine the snooped data to look for a single or a certain sequence of processor executions that violates the integrity of the host system.

In summary, this paper's contribution includes the following:

First, we present the prototypes of Vigilare system, each of which is designed to monitor a Linux kernel running on either the Gaisler grlib-based SoC or an ARM-based SoC. To the best of our knowledge [15], our Vigilare is the first hardware integrity monitor that implements the ability to snoop the bus traffic to perform an integrity analysis of an operating system kernel. Although most of the hardware-based approaches were based on snapshot analysis, one approach employed event-driven integrity monitoring [4], which has some similarities to snoop-based monitoring. However, it was a hypervisor-based approach which is not a perfectly safe execution environment as noted in the text.

Second, we propose an extended design of our prototype for the grlib-based SoC. This extension allows Vigilare system not only to detect malicious modifications to the host system main memory, which was supported by the first prototype [16], but also to prevent permanent damage to the contents of the main memory from malicious modifications. Although this design inevitably increases the memory access latency, we could minimize the performance overhead by exploiting the pipelined nature of the host system bus.

Third, to illustrate and provide a better understanding of the benefits of our snooper-based approach in comparison with previous snapshot-based approaches, we also implemented a snapshot-based monitor for the grlib-based SoC and created a sample transient attack testing code with tunable parameters for the experimental testing. The sample attack code takes advantage of snapshot-based solutions and we show how Vigilare effectively deals with these types of attacks.

Lastly, we present our comparison evaluation study, reporting that the prototypes of Vigilare system detected all of the transient attacks on immutable regions without degrading the performance, whereas the snapshot-based monitor could not detect all of the attacks and induced considerable performance degradation with an average of 17.5 % for a 50ms snapshot interval in our tuned STREAM benchmark test. This is due to the performance overhead incurred by the memory access to acquire snapshots periodically, which also consumes memory bandwidth at the host. In contrast, the first prototype for the grlib-based SoC replicates the bus traffic of the host system using a dedicated hardware module such that snooping can be performed without consuming any memory bandwidth at the host system. The second prototype also incurs negligible performance overhead, as the increased latency can be hidden by pipelining. In addition, the prototype for the ARM-based SoC also successfully detected all transient attacks with negligible performance overhead, despite its weakness in computing power compared to its host system.

The rest of this paper is composed of the following: first we describe our assumptions and threat model in Section 2 and define the transient attack in Section 3. Also, we explain the Vigilare system in detail in Section 4. We provide more details about our prototypes in Section 5, and present the extended design in Section 6. We evaluate our implementations in Section 7, and discuss limitations and future works in Section 8, related works in Section 9 and conclude our paper in Section 10.

# 2 Assumptions and Threat Model

## 2.1 Assumptions

We assume that the host system is already compromised by an attacker, that is, the attacker has gained the administrator's privilege on the host system. In addition, we assume that the attacker has no physical access to the entire system therefore we can rule out the possibility of hardware modification. Thus, we

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



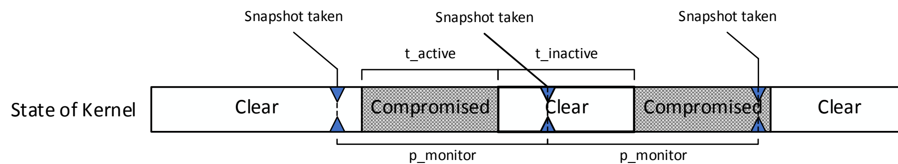


Fig. 1. This figure shows the behavior of transient attack. Transient attack compromises the kernel “transiently” for $t_{active}$ and remove its trace for $t_{inactive}$ to avoid being detected. Assuming that there exists a snapshot-based integrity monitor, it may detect the second pulse but fail to detect the first one. The transient attack with lower $t_{active}$ may have more chance to attack the host system successfully

are limiting the manipulation by the attacker in the realm of software; no modification should affect the operation of the Vigilare system hardware or any other hardware components.

### 2.2 Threat Model

The primary threat that the Vigilare system strives to mitigate is the kernel-level rootkits. As mentioned previously, we assume that the attackers have already gained control over the host system. In order to continuously steal data, and also to eavesdrop the users of the system, the attacker can conceal their footprints by manipulating the kernel, so that it can bypass the security mechanisms in the kernel or mislead the anti-malware solutions running on the host.

Another assumption is that the attackers might be aware of the presence of some kind of security monitor. The attackers aim to avoid detection with the best of their knowledge. One possible technique is to minimize and avoid an obvious and permanent modification to the kernel. By hiding the traces of malware left in the host memory, the attackers could lower the probability of getting detected. We define such malicious behavior *transient attacks* and the details will be covered in the next section.

## 3 Transient Attack

As mentioned before, previous kernel integrity monitoring schemes that utilize memory snapshots to find the traces of rootkits are vulnerable to transient attacks. On the other hand, Vigilare solves such shortcomings with its bus traffic monitoring architecture. In this section, we define the term transient attack, introduce some examples of such attacks, and discuss the challenges in detecting these transient attacks.

### 3.1 Definition

Transient attack is an attack whose traces do not incur persistent changes to the victim system. In such scenarios, the evidence of malicious system modification is visible for a short time period. In turn, detecting the modification becomes difficult. The term transient attack is defined rather broadly; any attack that does not leave permanent changes can be classified as transient attacks. The soft-timer based rootkit technique presented in the work by J. Wei et al. shows the aspects of transient kernel rootkit [17]. The rootkit designed by J. Wei et al. takes advantage of Linux *timer* data structure to convey the malicious code and execute it at the scheduled time. Since the code that had been contained in the timer is discarded shortly after its execution, its trace is not only difficult to locate but it also stays in the memory for a very short time. The exploitation of Linux soft-timer is a mere example, more sophisticated transient attacks that can nullify traditional rootkit detection solutions are here to stay.

### 3.2 Difficulties of Detecting Transient Attacks

In order to successfully detect transient attacks, the detection system needs to operate on an event-triggered mechanism; snapshot analysis or other periodic checks are likely to miss out the events in between the snapshots. Imagine a snapshot-based integrity monitor was launched to detect the transient attack model shown in Figure 1. If the author of the kernel rootkit can properly adjust the duration of the attack $t_{active}$ and the time of dormancy $t_{inactive}$, he could completely evade the snapshot-based monitors; by staying dormant at the time of memory snapshot and becoming active in between the snapshots, the rootkit can capably fool the snapshot-based monitor.

A possible temporary solution to the limitation of the snapshot-based approach would be to increase the rate of memory snapshot-taking or to randomize the snapshot interval. Frequent memory snapshots will, however, inevitably impose a high performance overhead on the host system. Also, random snapshot timings may not properly represent the system status and will produce either snapshots with little differences or snapshots with a long time interval in between. Therefore, we conclude that snapshot-based integrity monitors are simply not apt for detection of transient attacks. That is, an event-triggered solution is essential to cope with transient kernel attacks.

## 4 Vigilare System Requirements

Vigilare is designed to collect the data stream in the host system’s memory bus to overcome the limitations of memory-snapshot inspection. Figure 2 shows high level design of the Vigilare system. The Vigilare system is mainly composed of two components: Snooper and Verifier. Snooper is a hardware component which collects the traffic and transfers it to Verifier. Verifier is a compact computer system optimized to analyze the data in order to determine the integrity of the host system. In our prototype, Verifier is placed along with Snooper for simplicity and performance. In this section, we describe several requirements that the

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



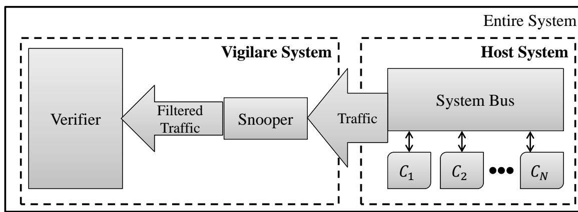


Fig. 2. This is a high-level design of the Vigilare system. $C_1$ through $C_N$ indicates the hardware components of the host system which connects each other via the system bus, such as main processor, memory controller, or network interface. The Vigilare system includes a Snooper that has hardware connections to the system bus of the host system to be monitored. Verifier analyzes the filtered traffic that Snooper provides.

Vigilare system needs to meet, so that it does not fail to detect important kernel status changes that appear in the bus traffic.

### 4.1 Selective Bus-traffic Collection and Sufficient Computing Power

As Verifier in general needs to analyze the filtered collection of bus traffic that Snooper provides, a bus bandwidth that is higher than Vigilare's computing speed can be problematic. The AHB (Advanced High-performance Bus) included in AMBA 2 (Advanced Microcontroller Bus Architecture) [18], which was used in the host system of our first prototype is a good example of such problem. Every cycle, 4 byte address and 4 byte data is transferred to memory controller through the bus along with a few more bits of bus-specific signals. Thus, we cannot process the stream of bus traffic in one cycle with a general purpose 32 bit machine. Therefore, Snooper must be designed with a selective bus-traffic collection algorithm; it should recognize only meaningful information while truncating other unnecessary traffic data flow.

The required time to process a filtered collection of traffic is also related to the computing power of Verifier. The more computing power Verifier has, the less time would be required to process the same collection of traffic. Verifier's computing power must be predetermined in the design process, so that it provides just enough processing power yet does not introduce excessive power consumption.

### 4.2 Handling Bursty Traffic

Just filtering out some traffic may not allow sufficient time to process all collections of traffic to Verifier. Filtering may reduce the processing load imposed on Verifier processor and enables Vigilare to cooperate with high bandwidth systems. However, even selective filtering does not guarantee that the rate of bus traffic collected is steady and expectable. That is, a deluge of meaningful information may coincidentally congregate within a short period of time, overwhelming the computing power of Verifier.

The most intuitive workaround would be to add a FIFO (first-in first-out) queue to Snooper to address the problem. Unfortunately, it does not effectively remedy the problem. First of all, implementing a FIFO for the specific architecture will inevitably bring up the hardware cost. In addition, it is rather difficult to estimate the proper size of FIFO and we cannot afford to discard the critical information when FIFO becomes full.

A better approach is to build a more abstract, interpretable data from the raw bus-traffic data in Snooper. It would require more logic implementation to make Snooper more complex. However, it would be much more efficient than simply increasing FIFO queue or equipping Verifier with powerful processor, since Snooper can filter and summarize the bus traffic so that the Verifier can avoid the handling of unrelated bursty traffic.

### 4.3 Integrity of the Vigilare System

Hardware-based integrity monitor has its advantage in the independence from the host system. Since it does not rely on the core functionalities of the host system kernel, the integrity of the host system does not affect the integrity of the Vigilare system. To further augment this strength of Vigilare, the memory interface of the Vigilare system and interrupt handling of Verifier should be designed with considerations for independence.

The memory of the Vigilare system contains all the programs and data used by the Vigilare system. The host system must not be able to access this memory in any way. One way of fulfiling this requirement is the use of separate memory for the Vigilare system, which makes the Vigilare memory become physically tamper-free. Alternatively, we may implement a memory region controller that specifically drops all memory operation requests from the host system. This option may reduce the cost of hardware implementation compared to that of building a completely separate memory for Vigilare.

The interrupt handling in Verifier can be a factor that undermines the independence from the host system. Thus, any circumstances that might trigger interrupts to Verifier should be carefully designed. More specifically, the peripherals controlled by host system should have no or limited ways of introducing interrupts to the Vigilare system.

## 5 Prototype Design

In this section, we describe our prototypes that we used for evaluating snoop-based monitoring. After describing the immutable regions of Linux kernel which our prototypes monitor as background information, we describe the details of the two prototypes of the Vigilare system: *SnoopMon* and *SnoopMon-A*, each of which investigates the integrity of immutable regions of the Linux kernel running on the grlib-based system and the ARM-based system repectively.

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



### 5.1 Immutable Regions of Linux Kernel

We define immutable regions as regions that are critical to the integrity of the operating system such that any modifications of the regions are deemed malicious. Protecting the integrity of immutable regions should be the highest priority, as any modifications of immutable regions in an attacker's favor would be the most detrimental type of modification to the system. The immutable kernel region constitutes critical components in the OS, and any compromise of this region would seriously affect all applications running on top of the OS. For instance, all countermeasures for detecting control-flow hijack attacks must be able to detect manipulations of immutable regions. Attackers can otherwise easily perform attacks by manipulating immutable regions. Therefore, our prototype of SnoopMon focuses on monitoring the immutable regions of the kernel.

As the target of integrity monitoring, we included kernel code region, system call table and *Interrupt Descriptor Table (IDT)*. Kernel code region is the most obvious example of immutable regions; the basic functionalities of the kernel must not be modified after the bootstrap, in other words, during the runtime. The system call table is another good example of immutable regions, as hijacking the kernel's system call often serves as an efficient way to control the kernel in the favor of an attacker. Modifying the system call table of the Linux kernel is a popular way to intercept the execution flow of the victimized system. The Linux system call table takes a form of an array of pointers. Each entry in the table points to a corresponding system calls such as sys_read, sys_write, and many more. The adversary could effortlessly hijack these system calls by inserting a function between the syscall table and the actual system call handlers. Most user mode applications as well as kernel mode ones, rely on the basic system calls to communicate with file system, networking, process information, and other functionalities. Therefore, taking control of the system call table enables one to control the entire kernel from the bottom. IDT is also an important immutable region as a critical gateway that kernel system calls pass through. By subverting such a low level system call invocation procedure, it is possible to hijack the system calls before even reaching the system call table.

### 5.2 Physical Addresses of Immutable Regions

As stated in [1], the virtual memory space used by the Linux generates semantic gap between the host system and the external monitor; independent monitors like Vigilare has no access to the paging files that manages the mapping between the virtual address space and physical memory address. Moreover, the operating system's paging mechanism often swaps out less frequently used pages to hard disk space.

However, the location in the kernel memory space

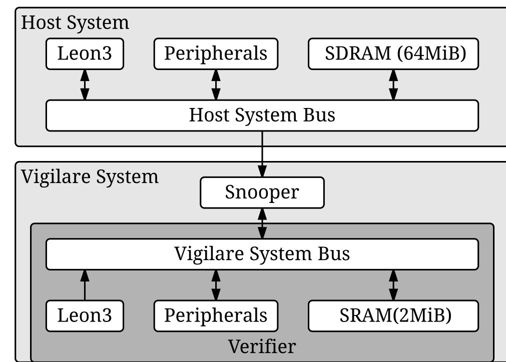


Fig. 3. This diagram shows the architecture of our prototype with SnoopMon. SnoopMon has its own memory and peripherals for its independence from host system. It snoops host system bus traffic with Snooper. Snooper delivers traffic that indicates write attempts to immutable regions of host system kernel. The peripherals of SnoopMon include a debug interface for out-of-band reporting.

in which the static region of kernel resides, is determined at boot time. Thus, we can reliably locate and monitor the important symbols within the static region. The virtual address of the kernel text is found in "/boot/System.map" file, which lists a numerous symbols used by the kernel; it is a look-up table that contains virtual addresses of symbol names. The symbol _text and _etext signifies the start and the end of the kernel's text section. The physical addresses of the text section can be calculated by adding the virtual addresses and a certain offset, which remain constant during runtime. The physical addresses of syscall table and IDT are also determined at compile time, and thus we can make use of them for monitoring.

### 5.3 SnoopMon

SnoopMon is the first prototype of the Vigilare system that is designed to monitor a Linux kernel for the grlib-based system. Specifically, it uses the Leon3 processor as the main processor which is a 32-bit processor [19] based on SPARC V8 architecture [20] provided by Gaisler. It is designed for embedded software with low complexity and low power consumption. The Leon3 processor has seven stage pipeline with Harvard architecture, 16KB instruction cache, 16KB data cache, and runs at 50MHz in our prototype SoC. The system uses 64MB SDRAM as a main memory and has some peripherals for debugging. It runs the Snapgear Linux [21] which is an embedded Linux customized for the processor and runs kernel version of 2.6.21.1.

Figure 3 shows the structure of SnoopMon, which is basically a separate computer system that consists of one Leon3 processor, Snooper, 2MB SRAM, several peripherals, and the bus that interconnects them. This configuration makes the memory and the memory interface of the Vigilare system independent from the host system. Moreover, the host system cannot access

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



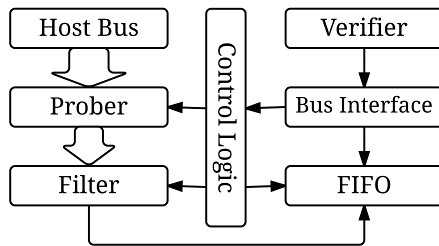


Fig. 4. This is a detailed description of Snooper. Verifier controls Snooper with the AHB slave interface and control logic. Prober drops inessential signals in the incoming bus traffic to reduce the bandwidth of the remaining logics in Snooper. Key logic to filter the traffic is implemented between Register_1 and Register_2. In SnoopMon, Filter_Proto is the key logic that compares the address field of the traffic with preset information pertaining to the immutable region. Two registers enhance the modularity of Snooper and make the length of the critical path shorter such that Snooper can operate at a higher frequency.

any peripheral of the Vigilare system and is also incapable of triggering interrupts to the Verifier. Therefore the Vigilare system design meets the requirement in Section 4.3.

To detect modifications to the immutable region that we explained in Section 5.1, SnoopMon specifically captures any write operation on those intervals of addresses, using its Snooper. As shown in Figure 4, the datapath of Snooper is composed of three modules: *prober*, *filter*, and *FIFO*. Prober drops inessential signals to reduce the bandwidth of remaining modules. AHB protocol defines many signals, but Vigilare system may not need all those signals for monitoring. For example, SnoopMon only needs HADDR and HWRITE signals among all the signals that AHB protocol defines. HADDR indicates target address of the traffic, and HWRITE is '1' if and only if the traffic is to write. Filter is what determines whether to pass the traffic or not, and the filter for our SnoopMon compares HADDR with boundary addresses of the immutable region and checks whether HWRITE is '1' or '0'. FIFO queue is the module to handle bursty traffic. FIFO queue stores the traffic that filter passed, and drop the traffic once Verifier pops it. It is worth noting that in this prototype, Verifier does not need to additionally examine the traffic from Snooper since only the write attempt to the immutable regions can pass the filtering of Snooper, all of which should be considered malicious. However, Verifiler of this prototype is still required as a kernel-independent entity that controls Snooper to monitor the immutable regions.

Although Snooper is capable of monitoring the bus traffic to the host main memory, access to memory contents that are in cache do not generate observable traffic on the bus. The bus traffic that indicates read attempts will be generated only when cache-miss occurs. On the other hand, write traffic is generated depending on the type of cache: write-through cache or write-back cache. In case of write-through cache, all write attempts by the processor may generate the corresponding packets on the bus. In case of write-back cache, the write attempts from the processor may not be seen immediately on the bus, because the write-back cache does not commit all the memory updates immediately to the memory. Thus, it is plausible to devise a transient attack that can live on write-back cache before the updated cache contents are flushed to the memory bus. However, predicting the time of the write-back cache flushing is not trivial, so implementing such a rootkit would be nearly impossible. In addition, attacks modifying the immutable regions cannot adopt the scheme to evade a snoop-based monitor since. Even if an attacker successfully performed the attack, it generates dirty bits in cache lines of the regions, and it will be written back. Thus we can detect the attack by snooping the write traffic to the memory. Most caches write back dirty bits even when it is restored to original value, so any cache attack on immutable region generates corresponding write traffic on the links between caches and memories, where SnoopMon is watching. Simply enforcing write-through policy on the host caches would also be an alternative solution.

### 5.4 SnoopMon-A

We implemented another prototype of the Vigilare system in order to demonstrate more clearly the efficacy of our scheme in monitoring a host system that is either equipped with a write-back cache or run faster than the monitor. Using a development board with a *Zynq all programmable SoC* [22], we were able to implement SnoopMon-A, which monitors a Linux kernel 3.8.0 running on a system with an ARM Cortex-A9 processor as its main processor. As the host system is equipped with separate 32KB instruction and data caches as well as an 512KB write-back L2 cache, SnoopMon-A would show if our scheme suffers from the write-back caches or not. In addition, the processor operates at 666MHz and connected to an 512MB DDR2 SDRAM through a bus that operates at 120MHz, while SnoopMon-A operates at 20MHz and uses 16KB on-chip memory.

As SnoopMon-A operates at a frequency lower than that of the bus, adopting the design of the Snooper for the first prototype would cause the monitor to miss some traffic. In order to overcome the frequency difference, Snooper for SnoopMon-A has an advanced prober, which operates at the same frequency as the host bus and an asynchronous FIFO between the prober and the filter that bridges the clock domain. Unlike the one for SnoopMon, the prober for SnoopMon-A is designed to drop most of the traffic without hindering its detection capability by taking advantage of line-based nature of memory accesses.

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



As the host system uses a write-back cache, most write accesses from the host processor are generated with a granularity of cache line, which is 64Bytes in case of the host system. Consequently, snooping and processing the address of only one word among the eight words of a cache line, Snooper can infer the other seven addresses. Following this principle, the prober passes only one traffic among eight traffics if they are in the same cache line so that SnoopMon-A operating 20MHz can handle the traffic of the bus running at 120MHz.

## 6 Protecting the Kernel from Permanent Damage

In this section, we explain the motivation behind protecting the OS kernel from permanent damage and introduce the third prototype, which enables such protection by snooping and blocking bus traffic. We say that an OS kernel is permanently damaged if any malicious modification is committed to the main memory such that we can no longer find the clear kernel in the system. Upon the detection of malicious kernel modifications, Vigilare system needs to restore the kernel to a clear one as soon as possible. If this does not occur, the compromised kernel will continue to run as the attacker intended. A straightforward action of removing the effects of malicious modifications on the kernel is to reboot the kernel. If the host system provides a secure boot [13], [14] and the system has a set of untampered boot images, the reboot process always guarantees that the original, clean kernel image is loaded, thus repairing all the actions taken by the attack. This approach, initially, appears simple and effective, but simply rebooting the system without considering the consequences can entail several problems that should be noted. First, the system should have a kernel-independent means to protect the boot images from malicious modifications. As the attackers have the administrator's privilege, they are capable of modifying the images in the storage of the system. Although they cannot deceive the system to boot with a malicious boot image due to the secure boot, they still can destroy the boot image so that the system cannot reboot until the user somehow restore the boot image. In addition, an immediate reboot of the system could result in the loss of all unsaved data and statuses of all applications, as there are a number of applications running on the kernel at runtime. If this is not desirable, the reboot should be postponed until all the data and statuses are restored, which will allow the attackers to have longer period of time to achieve their goal. Thus, Vigilare needs to repair the kernel without rebooting.

We are able to achieve the goal by preventing attackers from writing to the system main memory. Once the kernel in the main memory becomes corrupted, there is no way to restore the compromised

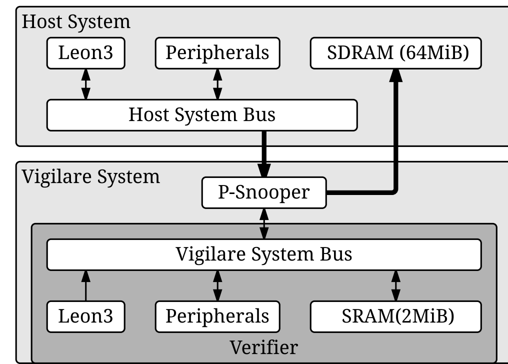


Fig. 5. This shows the SoC including our third prototype of Vigilare System, P-SnoopMon. Unlike Snooper in SnoopMon, P-Snooper intercepts bus traffic between the host system bus and memory controller, and blocks malicious traffic.

kernel to its original state, unless we duplicated all the previous values as backup. In contrast, any software-driven attacks can only corrupt kernel contents in the processor caches if the main memory is protected from such modifications. In this case, we can repair a compromised kernel in the processor caches by flushing the corresponding cache line. Because compromised cache lines are flushed without committing their changes to the main memory, the host system will continue to run a clear kernel in the main memory. Both SnoopMon and SnapMon cannot prevent this permanent damage to the kernel. SnapMon recognizes malicious modifications from a snapshot of the host system main memory after the kernel in the main memory is corrupted. SnoopMon may have a chance to detect a modification before the memory contents are changed, but it does not have any means of preventing the changes.

We designed and implemented P-SnoopMon, which not only detects all transient or persistent malicious modifications to kernel immutable regions as SnoopMon does, but also prevents all malicious modifications to the regions in the main memory of the host system. The latter allows P-SnoopMon to protect the host system kernel from permanent damage such that it can repair the compromised kernel without rebooting the system. For host systems with write-back caches, P-SnoopMon repairs compromised kernel image in the caches by flushing them and making use of the protected kernel image in the main memory. For host systems with write-through caches, P-SnoopMon even prevents rootkit attacks on immutable regions without repairing.

Figures 5 and 6 show the architecture of P-SnoopMon, which is composed of Verifier and *P-Snooper*. Unlike SnoopMon, P-SnoopMon not only detects but also blocks write traffic between the host system bus and the main memory. When the host system bus sends write traffic to P-Snooper, P-Snooper Datapath blocks the traffic and waits until P-Snooper Controller validates the traffic, which takes one cy-

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



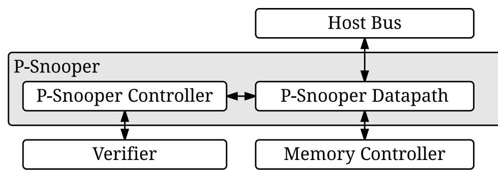


Fig. 6. P-Snooper is composed of two components: P-Snooper Controller and P-Snooper Datapath. The latter passes write traffic to the memory controller only when P-Snooper Controller validates the traffic. P-Snooper controller includes the part of Snooper shown in Figure 4 except the prober of P-Snooper controller, which is connected to P-Snooper Datapath instead of the host system.

cle. This capability allows P-SnoopMon to prevent kernel rootkit attacks from damaging the kernel permanently. In addition, as the host system of our SoC prototype has a write-through cache, the P-SnoopMon prototype prevents rootkit attacks on the immutable regions of the kernel, as described in Section 5.1. Although P-Snooper inevitably increases the memory access latency, this latency is much smaller than that of the main memory. We present a detailed discussion of the performance overhead in Section 7.3.2.

P-SnoopMon is also a separate computer system that is independent of the host system kernel. As shown in Figure 5, the detail of the Verifier is identical to that of SnoopMon. Although P-Snooper Datapath has an AHB slave interface which is connected to the host system bus, it is not controlled by the host system. P-Snooper controller, which is controlled by Verifier, controls the P-Snooper Datapath. P-SnoopMon Controller snoops bus traffic at P-Snooper Datapath to validate the traffic, and generate corresponding control signals for the P-Snooper Datapath. Given that P-Snooper is exclusively controlled by the Verifier and is independent of the host system, the P-Snooper design also meets the requirement described in Section 4.3.

# 7 EVALUATION

In this section we present the results of our experiments using the two prototypes of the Vigilare system (SnoopMon, and SnoopMon-A) and the *SnapMon*. First, we compare the snoop-based monitoring and snapshot-based monitoring in terms of performance overhead and the efficacy in detecting transient attacks, using SnoopMon and P-SnoopMon. Then we provide our evaluation study on effectiveness of the snoop-based monitoring for a powerful host system and more attacks with SnoopMon-A.

## 7.1 Comparison with snapshot-based monitoring

To better compare our snoop-based monitoring scheme with the snapshot-based method, we implemented a snapshot-based integrity monitor called SnapMon, following the design of such monitors represented by Copilot [1]. From SnoopMon, we implemented the hardware of SnapMon by replacing the Snooper with a DMA-capable memory interface

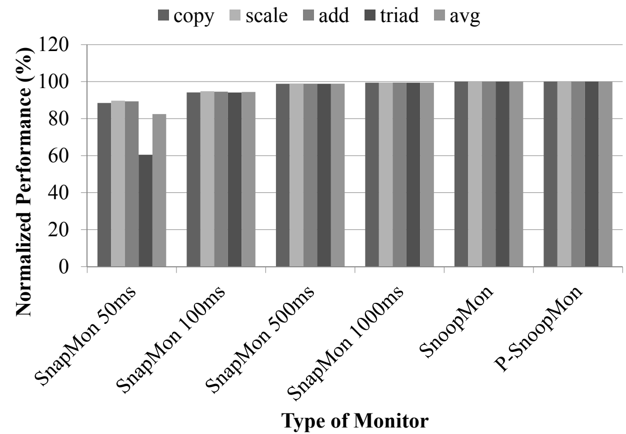


Fig. 7. Performance degradation due to each monitor. Copy, scale, add, and trial are subbenchmarks of STREAM. SnapMon with shorter snapshot interval (i.e., increased snapshot frequency) degrades performance of the host more.

such that SnapMon uses DMA to get the snapshot of the immutable regions of host system's kernel periodically. SnapMon calculates the hash value for each kernel region snapshot, and compares it against the pre-calculated hash value of the unmodified immutable region. In order to implement the monitoring functionality on the hardware, we first tried to use the processor for hashing the contents and some memory spaces to store snapshots. The memory we used for SnoopMon was sufficient for SnapMon since the size of immutable region in kernel was less than 1MB. However, the processing time for the hashing was longer than we had expected; it took about 5 seconds for each hash calculation for a snapshot. To reduce this, we included a hash accelerator in SnapMon to shorten the processing time to a reasonable level – approximately 1.3ms for each snapshot hashing.

We performed two experiments to compare the performance degradation and the efficacy in detecting transient attacks of each scheme. In case of SnapMon, the interval of snapshots significantly affects both performance degradation and ability to detect transient attacks. We varied SnapMon's interval to observe its effect on both detection rates and performance degradation. SnoopMon and P-SnoopMon do not have any parameter that possibly affects the result of experiments so we used only one type of SnoopMon and P-SnoopMon.

### 7.1.1 Performance Degradation

STREAM benchmark is widely used for measuring the memory bandwidth of a computer system. We used a tuned version of STREAM benchmark [23] to measure the performance degradations imposed on the host system. The original version uses double precision numbers for measuring the bandwidth but the Leon3 processor does not have floating point units. Thus, we tuned it to use integer numbers for measuring.

We let the tuned STREAM benchmark run on host system while each monitor is running. We averaged

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



1000 experiments to acquire more accurate results. As Figure 7 shows, SnoopMon and P-SnoopMon do not causes a discernable performance degradation, while SnapMon with shorter intervals degrades the performance significantly. For instance, performance degradation due to SnapMon with 50ms interval was measured to be 17.5% on average, 10% in the best case and about 40% in the worst case. If the snapshot interval is greater than 1 second, we have less performance degradation, 0.5% on average. This performance degradation was measured with the host system as-is after the boot, meaning that our benchmark tool is the only resource hungry process in the system. In addition, the performance degradation due to SnapMon also depends on the size of the immutable region, as it determines the size of snapshots to be taken and be transferred over. Note that both SnoopMon and SnapMon monitor the immutable regions that has the size of about 1 MB. As the Linux we used is tuned for embedded systems, the size of immutable regions may have been smaller than the kernels for desktops or modern smartphones.

#### 7.1.2 Transient Attack

To compare the efficacy of the monitors in detecting transient attacks, we implemented a rootkit example that meets the definition of transient attack in Section 3. The rootkit modifies the system call function pointers in *sys_call_table* as traditional Linux kernel rootkits in the wild, but restores the table after $t_{active}$, to avoid being detected by a snapshot-based monitor, as shown in Figure 1. While the figure illustrates a transient attack that performs hooking once again after $t_{inactive}$, our rootkit example has been designed to generate only one pulse of the attack so that we can measure the probability of detecting one pulse of the attack depending on $t_{active}$. Specifically, we generated 500 pulses and measured how many of them were detected by each monitor.

Figure 8 shows the results of the experiments. SnoopMon and P-SnoopMon detect all the pulses of attacks for all $t_{active}$, while SnapMon misses many of the transient attacks. Furthermore, P-SnoopMon also prevents all attacks that it detects, as mentioned in Section 6. SnapMon with 50ms snapshot interval detects all the pulses which have $t_{active}$ greater than 50ms, but SnapMon with 1000ms snapshot interval cannot detect more than 5% of the pulses when $t_{active}$ is less than 50ms. The results show that it is necessary to increase the snapshot frequency significantly for SnapMon to reliably detect transient attacks.

### 7.2 Effectiveness of Snoop-based Monitoring

In addition to the comparison study with the snapshot-based scheme, we have investigated the effectiveness of snoop-based monitoring through our additional prototype implemented on the ARM architecture, which we call SnoopMon-A. As we expected, SnoopMon-A caused negligible performance overhead regardless of the size of monitored regions, as well as successfully examined all write attempts to the immutable regions. We used the STREAM bench to measure the performance overhead, and checked if the asynchronous FIFO overflows or not, to make sure that Snooper receives every traffic representing a cache line eviction.

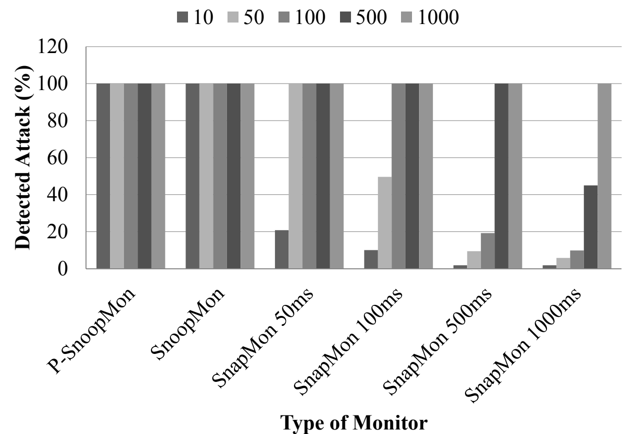


Fig. 8. Ratio of detected attacks for each monitor. Numbers in legend indicates $t_{active}$ of each pulse in the attacks in millisecond. This shows SnoopMon detects all the attacks while SnapMon cannot.

To further explore the efficacy of our scheme, we implemented three more rootkit examples in addition to the synthetic attack that we have described before, targeting the ARM-based system. One is another example of transient attack, which effectively hooks the system call table permanently for a particular process. The attack, which we call *per-process hooking*, manipulates the saved context information of the target process which is not currently running but in the scheduler's queue, in order to mislead the process to execute the *hooking payload* that performs system call hooking. To restore the system call table when the process is scheduled out, the payload also manipulates the kernel code such that the process invokes the *unhooking payload*, which restores both corrupted code and the system call table before switching to another process. This attack is transient in the sense that it only temporarily manipulates the immutable regions of the kernel, but effectively persistent to the target process as the hooking works for the process always. As this attack also modifies the immutable regions, SnoopMon-A could detect the attack when the target process is scheduled.

The other two rootkit examples are implemented to show the effectiveness of the snoop-based monitoring against the real-world rootkits, since all source code available rootkits that we have analyzed were designed for x86 systems. Because most of the rootkits have similar behavioral characteristic as shown in Table 1, we have implemented two rootkit examples for our host system following the representative behavior of them. One manipulates kernel code region

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



TABLE 1
Key behaviors of ten Linux kernel rootkits

| Name | Target Object | Object Type |
|---|---|---|
| Adore | sys_call_table | Immutable |
| Adore-NG 0.41 | inode→i_ops | Mutable |
| | task_struct→{flags,uid,...} | Mutable |
| | module→list | Mutable |
| Knark 2.4.3 | sys_call_table | Immutable |
| | proc_dir_entry | Mutable |
| | task_struct→flags | Mutable |
| | module→list | Mutable |
| KIS 0.9 | sys_call_table | Immutable |
| | proc_dir_entry | Mutable |
| | tcp4_seq_fops | Mutable |
| | module→list | Mutable |
| EnyeLKM 1.3 | sysenter_entry | Immutable |
| | module→list | Mutable |
| hideme.vfs | sys_getdents64 | Immutable |
| | proc_root_operations | Mutable |
| override | sys_call_table | Immutable |
| Synapsys-0.4 | sys_call_table | Immutable |
| fuuld | task_struct | Mutable |
| | sys_call_table | Immutable |
| net3 | nf_hook_ops | Mutable |

to intercept system calls, and the other modifies only mutable region to hijack the control-flow of the kernel. As we have expected, SnoopMon-A could detect the first example without difficulty, but not the second one as SnoopMon-A watches the modifications to the immutable regions only. (Note that the prevention of mutable region modificaion is beyond the scope of this paper as it is non-trivial task. For example, the value of mutable region changes over time and it is difficult to keep track of the genuine value to be restored in the event of an attack. )

### 7.3 Discussions

#### 7.3.1 Difference between SnapMon and SnoopMon

The comparison study shows that snapshot-based monitoring has trade-off between the performance degradation and the ability to detect transient attacks. With snapshot interval higher than 1 second, the SnapMon may cause little performance degradation but it misses out some transient attacks. Even the rootkit example with $t_{active}$ 500ms, SnapMon with snapshot interval 1 second could only detect about 50% of them. In order to detect most of transient attacks, we need to lower the snapshot interval of monitoring but it results in significant performance degradation of the host system. However, SnoopMon and P-SnoopMon detect all the transient attacks with little or no performance degradation by employing snoop-based monitoring.

#### 7.3.2 Performance Overhead of P-SnoopMon

As mentioned in Section 6, P-SnoopMon unavoidably increases the memory access latency, but the experimental results presented in Section 7.1.1 do not show any performance degradation. Figure 9 shows the reason why P-SnoopMon does not incur any visible performance overhead. Each block of the timelines in the figure indicates the abstract view of signals composing the traffic at each clock cycle. As shown at the beginning of the timelines, P-SnoopMon increases the memory access latency by one cycle, which is much smaller than the latency without P-snoopMon, latency_back in general. In our prototype, the value of latency_back is 12.

Furthermore, P-SnoopMon can pipeline a burst mode memory access by exploiting the pipelined nature of AHB protocol. As shown in Figure 9, the host system bus sends the second destination address, Address(1), before the first transaction is completed. If the latency_back is larger than 1, the additional memory latency due to P-SnoopMon can be hidden as shown in lower three timelines of the figure. P-SnoopMon can validate the next traffic while it waits for the response of the memory controller, since the value of latency_back is 12. Consequently, P-SnoopMon increases the latency of a burst mode memory access by only one cycle. Given that most of the memory accesses in the host system are performed in burst mode, P-SnoopMon causes negligible performance overhead as shown in the experimental result.

## 8 Limitations and Future Works

In this section, we discuss about some limitations, and future works.

### 8.1 Dynamic Kernel Region Manipulations

The most prominent trend in the latest kernel rootkit technique is the dynamic kernel region manipulation. In order to evade the common rootkit detection tools that easily detect obvious static kernel hooking, the forgers of kernel rootkits are looking aside to *DKOM (direct kernel object manipulation)*. By directly modifying dynamically created and removed kernel data structures that are used in core kernel operations, the attackers can manipulate the system status in a stealthy manner. In addition, ensuring the integrity of static immutable regions does not necessarily guarantee the integrity of the control flow of kernel [24].

Our next step in Vigilare project will be to design and implement a control flow integrity solution using our already implemented Snooper architecture. More specifically, a detection algorithm that senses each context-switching in the kernel space should be implemented to successfully detect control flow hijackings. Furthermore, a more elaborate semantic detection algorithm should be developed in order to detect malicious kernel manipulations in the mutable region.

### 8.2 Relocation Attack

Our snoop-based monitor prototypes cannot deal with relocation attacks. The relocation attack refers to moving the entire or parts of kernel to another location to stay out of the range of integrity monitoring. However, relocating, or copying of a large volume of the kernel code will inevitably produce an abnormal bus traffic pattern. Therefore, we expect that Vigilare

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



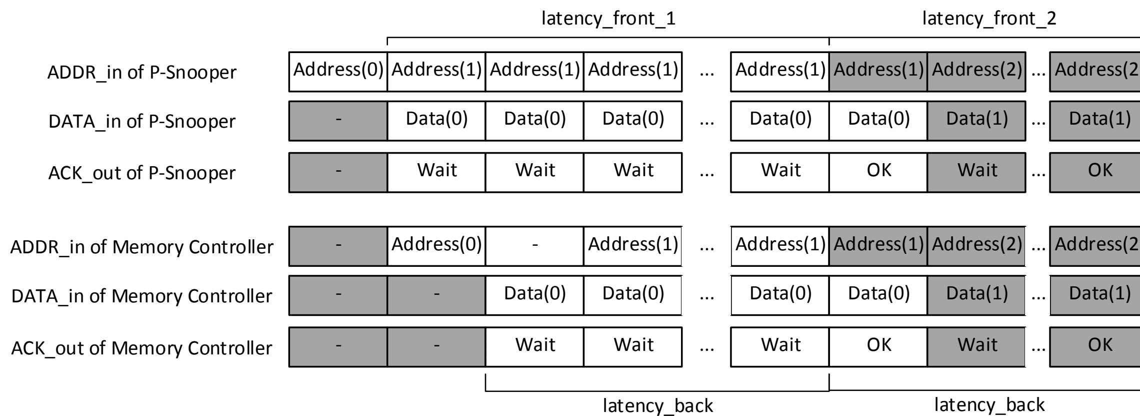


Fig. 9. These timelines show the effect of P-SnoopMon on memory access latency. The bright part indicates the first transaction of consecutive memory accesses. Latency_front_1 is the effective latency of the first memory access, and it is one cycle longer than latency_back, which would be the memory access latency of the host system without P-SnoopMon. However, the latencies of the other consecutive memory accesses are latency_front_2, which is the same latency_back

could be extended to detect such attacks with a prior knowledge of the existence of such attack patterns.

This assumption varies across different computer architectures. If the system uses Harvard architecture, it would be relatively easier to detect the relocation attacks. For the initial relocation, the attacker should read the kernel code as data, not as instruction. Since such behavior is quite unusual, we can detect it by analyzing the traffic. Even for the systems based on the Von Neumann architecture, the traffic pattern on the immutable regions under relocation attacks would be far from normal status. In both cases, snapshot-based monitoring would not be helpful since it gets only an instance of the system state, not the sequence of actions changing the states.

Another type of a relocation attack, called *Address Translation Redirection Attack* (ATRA) [25], has recently been published. The monitoring of the page table pointing register is indispensable for attacks on address translations, and thus it would require a specific hardware modification. For this reason, we regard that a ATRA defense scheme would be out of the scope of this paper. However, Addressing ATRA should be the most urgent future work for us.

### 8.3 Code Reuse Attacks

As the countermeasures against code injection attacks such as Data Execution Prevention (DEP) are employed in modern operating systems, many types of malware undertake a Code Reuse Attack (CRA) to exploit the vulnerabilities of target applications. CRA allows attackers to execute their own codes without injecting new code blocks or manipulating existing codes. Thus, CRAs work even when the execution of data and manipulation of codes are completely prevented. Rootkits can also perform CRAs to achieve their goals as shown in recent works [26], [27]. Although the proof-of-concept rootkit proposed in [27] overwrites system call table entries, which allows our prototypes to detect the rootkit, a conceivable rootkit may perform a CRA and avoid manipulating the immutable regions at the same time. Our prototype cannot detect this sort of attack, but countermeasures to these advanced attacks should also include solutions like our prototypes to mitigate simple but powerful attacks.

### 8.4 Privilege Escalation

While Vigilare concentrate on detection and prevention of kernel-level rootkits, the mitigation of *Privilege escalation attacks* are not discussed in this paper. We consider privilege escalation attacks as a preliminary stage that leads to our attack model. The attacker executes an arbitrary code (by injecting or reusing existing) or have obtained the root user account. Then, the attacker subverts the kernel code with rootkits to track system activities or steal important credentials. Vigilare focuses on preventing this perpetuation stage of the intrusion by ensuring the integrity of the kernel static regions.

In addition, detecting privilege escalation attack is a rather broad topic. A privilege escalation might be achieved through a vulnerability in the code that is running with a kernel privilege (i.e., the ring 0), alternatively, a process running with a root privilege (i.e., UID=0) could also be exploited to concede the privilege to the attacker. Either way, the topic of privilege escalation attack mitigation would include nearly the entire realm of software attacks and mitigation techniques – the difference is that the targeted code is running with the privilege targeted by the attackers.

## 9 Related Works

In this section, we explain previous approaches on protecting the integrity of an operating system kernel, which includes: kernel integrity monitors, rootkit detectors, and intrusion detection systems. The dilemma in designing such tools is that the security monitoring tool itself can be tampered with, if the malware operates on the same privilege level as that of the

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



monitors. To cope with this problem, many security researchers strive to make their security monitors independent from the system that is being monitored. A separated and tamper-free execution environment must be preceded before any advanced detection scheme. [15]

We can categorize prior works that aim to provide a solution to the problem, into two groups: hardware-based approaches and hypervisor-based approaches. We summarize these approaches for the rest of this section.

### 9.1 Hypervisor-based Approaches

Virtualization solutions, commonly called VMMs (Virtual Machine Monitors) or Hypervisors are widely used nowadays to efficiently distribute computing power among different types of needs. Since the hypervisor resides in between the hardware and the virtual machines, the hypervisor possesses the scope to manage and monitor the virtualized operating systems.

There has been quite a few works that take advantage of hypervisors for monitoring the security of the virtualized computers. One of the first in such works was Livewire [3] proposed by Garfinkel et al. Livewire proposed security monitor installed virtual machines. More ideas based on hypervisor has been proposed and implemented on popular hypervisors such as Xen [24], [28].

Although positioned underneath and separated from the virtual machines, it has been warned that the hypervisors can be also exploited with software vulnerabilities. Many vulnerabilities of Xen are already reported and amended [5], [6], [7], [8]. The discovery of hypervisor vulnerabilities might continue as the hypervisors are expanding in terms of code size and software complexity. This implies that the hypervisor might not be a safe independent execution environment, which is an imperative requirement for a security monitor.

There has been attempts to design minimal hypervisors for more secure execution environment for security monitoring [29], [30], [31], [32]. The idea is to include only essential software components to minimize the attack surface for software vulnerabilities. Some of such works used static analysis to ensure that their hypervisor is vulnerability-less.

Among such approaches, Secvisor [32] forces the CPU to execute only approved code in kernel mode by taking advantage of the Memory Management Unit (MMU) and the IO Memory Management Unit (IOMMU). This allows SecVisor to prevent unapproved codes from running in privileged mode, but limiting the use of mixed memory pages which contain both codes and data, which exist in numerous modern operating systems. It was necessary to modify the kernel linker script to remove the mixed code and data pages from the kernel.

In comparison with Secvisor, our prototypes cannot prevent maliciously injected code in data pages from being executed in privileged mode, but ours can monitor several important function pointers in immutable regions. Moreover, our prototypes can monitor an unmodified kernel regardless of the existence of mixed pages, and yet incur negligible performance overhead, whereas Secvisor cannot run an unmodified kernel and incurs greater performance degradation than the popular hypervisor software Xen.

Recently, Rhee et al [4] proposed an event-driven integrity monitor based on hypervisor. With event-driven nature, it can be considered as a hypervisor version of snoop-based monitoring. However, the security of the integrity monitor itself heavily relies on the premise that the hypervisors are vulnerability free. Besides, it reported non-negligent performance degradation.

### 9.2 Hardware-based Approaches

Another approach in implementing a kernel integrity monitor out of operating system is attaching an independent hardware component. The idea of securing operating system using SMP (Symmetric Multi-Processor) was first proposed by Hollingworth et al. [33]. Later, X.Zhang et al. proposed IDS (Intrusion Detection System) based on a coprocessor independent from the main processor [2]. Petroni et al. designed and implemented Copilot [1], which is a kernel runtime integrity monitor operating on a coprocessor PCI-card. More snapshot-based works followed after Copilot and inherited the limitations of snapshot-based mechanism presented in Copilot. [34], [35].

Intel also contributed to the trend, by presenting a hardware-based support snapshot-based rootkit detection called as DeepWatch [36]. J. Wang et al. designed HyperCheck [10] which is an integrity monitor for hypervisors based on a PCI card and the SMM (System Management Mode) [12]. A. M. Azab et al also proposed a framework called HyperSentry [9] for monitoring the integrity of hypervisors with their agent planted in the SMM. The critical drawback of using SMM for security monitoring is that all system activities must halt upon entering SMM. It implies the host system has to stop, every time the integrity monitor on SMM runs. DeepWatch and HyperCheck focused on building a safe execution environment but they both utilized memory snapshots for integrity verification.

In all, most of the hardware-based approaches use memory or register snapshots [1], [10] as the source of system status information. However, they are inapt for monitoring instant changes occur in the host system and thus vulnerable to advanced attacks such as transient attacks. HyperSentry [9] also uses the state of host system at certain points of time, when the independent auditor stops the host system and execute the agent. Thus, this can be considered as a snapshot-

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing



based monitor along with Copilot [1] and Hyper-Check [10], in the sense that they all use the periodically acquired status information. Our approach is fundamentally different from the previous snapshot-based approaches on hardware-based integrity monitors since our Vigilare is snoop-based monitor.

### 9.3 Snooping Bus Traffic

Snooping bus traffic is well known concept as shown in these two prior works. Clarke et al. [37] proposed to add special hardware between caches and external memories to monitor the integrity of external memory. The aim of this work is to ensure that the value read from an address is the same as the value last written to that address. It can defeat attacks to integrity of external memory, but cannot address rootkits nor monitor the integrity of operating system kernel, unlike Vigilare System.

BusMop [38] designed a snoop-based monitor which is similar to our SnoopMon, but the objective of BusMop is different from SnoopMon. BusMop is designed to monitor behavior of peripherals. Unlike BusMop, SnoopMon is to monitor the integrity of operating system kernel. To the best of our knowledge, Vigilare is the first snoop based approach to monitor OS kernel integrity while all of the previous approaches in this area were based on taking periodic snapshots.

## 10 Conclusions

In this paper, we proposed snoop-based monitoring, a novel scheme for monitoring the integrity of operating system kernels. We investigated several requirements on implementing our scheme and designed the Vigilare system and its snoop-based monitoring. We focused on contributing improvements over the previous approaches in two main aspects: detecting transient attacks and minimizing performance degradation. To draw the contrast between Vigilare and snapshot-based integrity monitoring, we implemented SnapMon which represents snapshot-based architecture. We pointed out that the snapshot-based integrity monitors are inherently vulnerable against transient attacks and presented our Vigilare system as a solution. In our experiment, we demonstrated that SnoopMon and P-SnoopMon are capable of effectively coping with transient attacks that violate the integrity of the immutable regions of the kernel, while snapshot-based approach had their limitations. In addition, P-SnoopMon also protects the host system kernel against permanent damage. We also investigated the performance impact on the host system using STREAM benchmark [23], and showed that SnoopMon, due to its independent hardware module for bus snooping, imposes no performance degradation on the host. P-SnoopMon also caused negligible performance overhead although it increases memory access latency. Snapshot-based integrity monitoring proved to be unsuitable for detecting transient attacks in general; it is inefficient because of the trade-off between detection rates and performance degradation; higher snapshot frequencies might improve the detection rates, but the performance suffers from the overused memory bandwidth. In all, Vigilare overcomes the limitation of snapshot-based integrity monitors with snoop-based architecture.

## References

[1] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot - a coprocessor-based kernel runtime integrity monitor,” in Proceedings of the 13th conference on USENIX Security Symposium - Volume 13, ser. SSYM’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 13–13.

[2] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, “Secure coprocessor-based intrusion detection,” in Proceedings of the 10th workshop on ACM SIGOPS European workshop, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 239–242.

[3] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in Proc. Network and Distributed Systems Security Symposium, February 2003.

[4] J. Rhee, R. Riley, D. Xu, and X. Jiang, “Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring,” in Availability, Reliability and Security, 2009. ARES ’09. International Conference on, march 2009, pp. 74 –81.

[5] Vmware : Vulnerability statistics. [Online]. Available: http://www.cvedetails.com/vendor/252/Vmware.html

[6] Vulnerability report: Vmware esx server 3.x. [Online]. Available: http://secunia.com/advisories/product/10757

[7] Vulnerability report: Xen 3.x. [Online]. Available: http://secunia.com/advisories/product/15863

[8] Xen : Security vulnerabilities. [Online]. Available: http://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html

[9] 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, ser. CCS ’10. New York, NY, USA: ACM, 2010, pp. 38–49.

[10] J. Wang, A. Stavrou, and A. Ghosh, “Hypercheck: A hardware-assisted integrity monitor,” in Recent Advances in Intrusion Detection, ser. Lecture Notes in Computer Science, S. Jha, R. Sommer, and C. Kreibich, Eds. Springer Berlin / Heidelberg, 2010, vol. 6307, pp. 158–177, 10.1007/978-3-642-15512-3-9.

[11] F. Zhang, J. Wang, K. Sun, and A. Stavrou, “Hypercheck: A hardware-assistedintegrity monitor,” Dependable and Secure Computing, IEEE Transactions on, vol. 11, no. 4, pp. 332–344, 2014.

[12] L. Duflot, D. Etiemble, and O. Grumelard, “Using cpu system management mode to circumvent operating system security functions,” in In Proceedings of the 7th CanSecWest conference, 2006.

[13] W. Arbaugh, D. Farber, and J. Smith, “A secure and reliable bootstrap architecture,” in Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on, 1997, pp. 65–71.

[14] B. Kauer, “Oslo: improving the security of trusted computing,” in Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, ser. SS’07. Berkeley, CA, USA: USENIX Association, 2007, pp. 16:1–16:9.

[15] J. Bickford, R. O’Hare, A. Baliga, V. Ganapathy, and L. Iftode, “Rootkits on smart phones: attacks, implications and opportunities,” in Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications, ser. HotMobile ’10. New York, NY, USA: ACM, 2010, pp. 49–54.

[16] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, “Vigilare: toward snoop-based kernel integrity monitor,” in Proceedings of the 2012 ACM conference on Computer and communications security, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 28–37.

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2015.2443803, IEEE Transactions on Dependable and Secure Computing




[17] J. Wei, B. Payne, J. Giffin, and C. Pu, "Soft-timer driven transient kernel control flow attacks and defense," in Computer Security Applications Conference, 2008. ACSAC 2008. Annual, dec. 2008, pp. 97 –107.

[18] AMBA$^{TM}$ Specification, ARM Limited, May 1999.

[19] GRLIB IP Core User's Manual, Aeroflex Gaisle, January 2012.

[20] The SPARC Architecture Manual, SPARC International Inc., 1992.

[21] D. Hellström, SnapGear Linux for LEON, Gaisler Research, November 2008.

[22] Xilinx, "Zynq-7000 all programmable soc technical reference manual," 2013.

[23] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25, Dec. 1995.

[24] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in Proceedings of the 14th ACM conference on Computer and communications security, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 103–115.

[25] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang, "Atra: Address translation redirection attack against hardware-based external monitors," in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014, pp. 167–178.

[26] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms." in USENIX Security Symposium, 2009, pp. 383–398.

[27] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert, "Persistent data-only malware: Function hooks without code," 2014.

[28] N. A. Quynh and Y. Takefuji, "A novel approach for a file-system integrity monitor tool of xen virtual machine," in Proceedings of the 2nd ACM symposium on Information, computer and communications security, ser. ASIACCS '07. New York, NY, USA: ACM, 2007, pp. 194–202.

[29] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in Proceedings of the nineteenth ACM symposium on Operating systems principles, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 193–206.

[30] K. Kaneda. Tiny virtual machine monitor. [Online]. Available: http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/

[31] R. Russell. Lguest: The simple x86 hypervisor. [Online]. Available: http://lguest.ozlabs.org/

[32] 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, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 335–350.

[33] D. Hollingworth and T. Redmond, "Enhancing operating system resistance to information warfare," in MILCOM 2000. 21st Century Military Communications Conference Proceedings, vol. 2, 2000, pp. 1037 –1041 vol.2.

[34] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in Computer Security Applications Conference, 2008. ACSAC 2008. Annual, dec. 2008, pp. 77 –86.

[35] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in Proceedings of the 15th conference on USENIX Security Symposium - Volume 15, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006.

[36] Y. Bulygin and D. Samyde, "Chipset based approach to detect virtualization malware a.k.a. deepwatch," in BlackHat USA, 2008.

[37] D. Clarke, G. E. Suh, B. Gassend, M. van Dijk, and S. Devadas, "Checking the integrity of a memory in a snooping-based symmetric multiprocessor (smp) system," MIT LCS memo-470, http://csg.csail.mit.edu/pubs/memos/Memo-470/smpMemoryMemo.pdf, Tech. Rep., 2004.

[38] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable cots-based real-time embedded systems," in Proceedings of the 2008 Real-Time Systems Symposium, ser. RTSS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 481–491.


PLACE PHOTO HERE

**Hyungon Moon** received B.S. degrees in Electrical Engineering and in Mathematical Science from Seoul National University, Korea, in 2010. He is currently working towards the Ph.D degree in electrical and computer engineering from Seoul National University. His research interests include improving operating systems and computer architectures for system security.

PLACE PHOTO HERE

**Hojoon Lee** received his B.S degree in Electrical and Computer Engineering from the University of Texas at Austin in 2010. He received his M.S degree in Information Security from KAIST in 2013 and he is continuing on for a Ph.D at the same institution (Graduate School of Information Security, KAIST). His research interests include trusted execution environment, hardware-based kernel integrity monitors, and virtual machine introspection.

PLACE PHOTO HERE

**Ingoo Heo** received a B.S. degree in Electrical Engineering from Seoul National University, Korea, in 2009. He is currently working towards the Ph.D degree in electrical and computer engineering from Seoul National University. His recent research interests are security hardware and data leak prevention using dynamic information flow tracking.

PLACE PHOTO HERE

**Kihwan Kim** received the B.S degree in KAIST in 2010 and M.S degree in Information Security from KAIST in 2013 He is pursuing his Ph.D at the same institution. He focuses on system security including kernel integrity monitors, remote attestation and security of controller systems.

PLACE PHOTO HERE

**Yunheung Paek** received the B.S. and M.S. degrees in computer engineering from the Seoul National University, Korea in 1988 and 1990, respectively. He received his Ph.D. degree in computer science from University of Illinois at Urbana-Champaign in 1997. Currently he is a professor at the department of electrical and computer engineering, Seoul National University, Korea. His research interests include system security with hardware and hypervisor, secure processor design against various types of threats, and encryption hardware. He is also working on mobile cloud computing and re-targetable compiler.

PLACE PHOTO HERE

**Brent Byunghoon Kang** is currently an associate professor at the GSIS (Graduate School of Information Security) at KAIST (Korea Advanced Institute of Science & Technology). Before KAIST, he has been with George Mason University as an associate professor in the Volgenau School of Engineering. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. He has been working on systems security area including OS kernel integrity monitor, trusted execution environment, hardware-assisted security, botnet malware defense, and DNS analytics. He is currently a member of the IEEE, the USENIX and the ACM.

1545-5971 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.