Vigilare: Toward Snoop-based Kernel Integrity Monitor

Hyungon Moon
Seoul National University
hgmoon@sor.snu.ac.kr

Hojoon Lee
Korea Advanced Institute of Science and Technology
hjlee228@kaist.ac.kr

Jihoon Lee
Seoul National University
jhlee@sor.snu.ac.kr

Kihwan Kim
Korea Advanced Institute of Science and Technology
kimgun@kaist.ac.kr

Yunheung Paek
Seoul National University
ypaek@snu.ac.kr

Brent Byunghoon Kang*
George Mason University
bkang5@gmu.edu

ABSTRACT

In this paper, 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. We implemented a prototype of the Vigilare system on Gaisler's grlib-based system-on-a-chip (SoC) by adding *Snooper* hardware connections module to the host system for bus snooping. To evaluate the benefit of snoop-based monitoring, we also implemented similar SoC with a snapshot-based monitor to be compared with. The Vigilare system detected all the transient attacks without 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.

Categories and Subject Descriptors

D.2.6 [**Operating Systems**]: Security and Protection—*invasive software*

General Terms

Security

Keywords

Transient Attack, Hardware-based Integrity Monitor, Kernel Integrity Monitor

*Corresponding author

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

1. INTRODUCTION

To protect the integrity of operating system kernels, many security researchers strive to make their security monitors independent from the host system that is being monitored. Recent efforts on this kernel integrity monitoring can be categorized into two groups: hardware based approaches [20, 30] and hypervisor based approaches [14, 24]. Recently, approaches based on hypervisors have gained popularity. However, as hypervisors are becoming more and more complex, hypervisors themselves are exposed to numerous software vulnerabilities [1, 2, 3, 4]. Several approaches [7, 28] 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 solutions to kernel integrity monitoring make use of snapshot analysis schemes; they are usually assisted by some type of hardware component that enables saving of the memory contents into a snapshot, and then perform an analysis to find the traces of a rootkit attack. HyperSentry [7], Copilot [20], and HyperCheck [28] 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 an inherent weakness because they only inspect the snapshots collected over a certain interval, missing the evanescent changes in between the intervals. 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.

Attackers can exploit this critical limitation of snapshot-based kernel integrity monitoring. If attackers know the existence of a snapshot-based integrity monitor and estimate the time 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 [7] 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 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 snapshot, and not detected. HyperSentry also noted it was not designed to address such transient attack.

As to detecting such attacks using snapshot-based approaches, raising the rate of snapshot-taking might increase the probability of detection. However, frequent snapshot-taking would inevitably introduce increased overhead to the host system. Randomizing the snapshot interval of the monitor can be another solution to defeat such deliberately designed transient attacks. Nonetheless, the detection rate would greatly depend on luck and not be consistent. If the transient attack is short-lived, not repeating its transient attacks, the chance of detection based on random snapshot interval would be low.

In this paper, we propose Vigilare, a snoop-based integrity monitoring scheme that overcomes the limitations of existing 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, 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. 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. This Vigilare system is composed of the following components. *Snooper* on the Vigilare system is connected to the system bus of the main system, and collects the contents of real-time bus traffic. Snooper 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 design of Vigilare system as a single SoC (System-on-a-Chip), with Linux as its main operating system, and our prototype implementation on the Gaisler grlib-based SoC by adding newly designed *Snooper* hardware connections module to the host system for bus snooping. To the best of our knowledge [9], our Vigilare is the first hardware integrity monitor that implements snooping capability of the bus traffic to perform integrity analysis of the operating system kernel. Although most of the hardware-based approaches were based on snapshot analysis, there was an approach that employed event-driven integrity monitoring [24], which has some similarities to the snoop-based monitoring. However, it was a hypervisor-based approach which is not a perfectly safe execution environment as we mentioned.

Second, to illustrate and better understand the benefit of our snooper-based approach in comparison with previous snapshot-based approaches, we also implemented a similar SoC with a snapshot-based monitor, and created a sample transient attack testing code with tunable parameters for the experiment testing. The sample attack code takes advantage of the snapshot-based solutions and we show how Vigilare effectively deals with the attack.

Third, we then present our comparison evaluation study, reporting that the Vigilare system detected all the transient attacks on immutable regions without performance degradation while the snapshot-based monitor could not detect all the attacks and induced considerable performance degradation with an average of 17.5 % for 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 Vigilare system replicates the bus traffic of the host system using dedicated hardware module, so the snooping can be performed without incurring any memory bandwidth consumption at the 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 evaluate our implementations in Section 6. We discuss limitations and future works in Section 7, related works in Section 8 and conclude our paper in Section 9.

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 are limiting the manipulation by the attacker in the realm of software; no modification should affect the operation of the Vigilare system SoC 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, and continue their attack while hiding the evidence of their intrusion. To achieve such goals, they are capable of modifying some parts of kernel. For instance, the attackers can install kernel rootkit that places hooks on critical system calls. Another assumption is that the attackers are 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 in previous sections, 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's 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 [29]. 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 Our Transient Attack Model and Examples

As a concrete example that we can test and observe, we devised a simple rootkit that exhibits transient characteristics. The rootkit acts similar to the traditional Linux kernel rootkits in the wild. Inspired by J. Wei et al.'s work, we implemented the rootkit to repeat modification and reversion in a fixed time interval using the Linux timer. Our example rootkit modifies the system call function pointers in *sys_call_table* to hijack the control flow of the system call. To be more specific on Linux system call hooking, the Linux system call table takes a form of an array of pointers. Each entry in the table points to a corresponding system call such as *sys_read*, *sys_write*, and many more. The adversary could effortlessly hijack these system calls by inserting a function between the *sys_call_table* and the actual system call. The example rootkit simply performs the old-fashioned system call hooking, but the timer-triggered operation allows us to illustrate the transient rootkit characteristics.

3.3 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. In the experiment, we designed our rootkit to have $t_{inactive}$ infinite, in order to measure the abilities of monitors to detect single pulse of attack.

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.

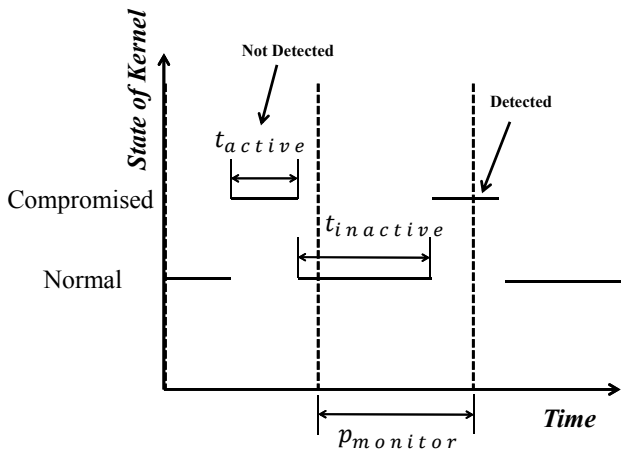

Figure 1: This figure shows the behavior of transient attack. Transient attack compromises the kernel "transiently" for t_{active} and remove its trace for $t_{inactive}$ to avoid being detected. Assuming that there exist a snapshot-based integrity monitor, it may detect the second pulse but fail to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

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. There are several requirements that the Vigilare system needs to meet, so that it does not fail to detect important kernel status changes that appear in the bus traffic. In this section, we describe these requirements of the Vigilare system.

4.1 Selective Bus-traffic Collection and Sufficient Computing Power

In the Vigilare system, Verifier analyzes 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)

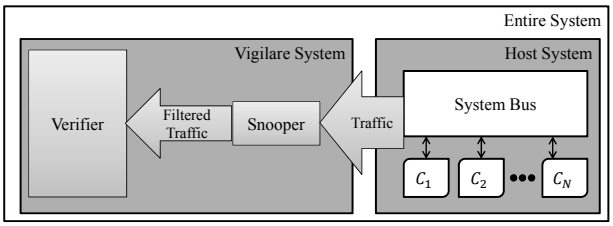

Figure 2: This is a high-level design of the Vigilare system. C_1 through C_N indicates the hardware components of the host system which connects each other via the system bus, such as main processor, memory controller, or network interface. The Vigilare system includes a Snooper that has hardware connections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

included in AMBA 2 (Advanced Microcontroller Bus Architecture) [6], which was used in the host system of our 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 expect-able. 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 were 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. There are two possible ways of meeting this requirement. The use of separate memory for the Vigilare system is the first option. By using a separate memory and memory controller inaccessible from the host system, the Vigilare memory becomes physically tamper-free. The second option is to implement a memory region controller which specifically drops all memory operation requests from the host system. The second 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. We designed and implemented two SoC prototypes: SnoopMon and SnapMon. Each SoC consists of a host system and an integrity monitor. SnapMon is an example of a snapshot-based monitor, and SnoopMon, which is a prototype of the Vigilare system, exhibits a snoop-based integrity monitoring scheme. Each monitor investigates the integrity of immutable regions of the Linux kernel. We first explain the host system and the immutable regions of Linux kernel as background information, and then we describe the details of SnoopMon and SnapMon designs.

5.1 Host System

We used the Leon3 processor as a host system's main processor which is a 32-bit processor [5] based on SPARC V8 architecture [27] 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 and runs at 50MHz in our prototype SoC. It has 16KB instruction cache and 16KB data cache. The system uses 64MB SDRAM as a main memory and has some peripherals for debugging. It runs the Snapgear Linux [15] which is an embedded Linux customized for the processor and runs kernel version of 2.6.21.1. We monitored the integrity of the Linux kernel with two monitors: SnoopMon and SnapMon. We provide more details about the specific target of monitoring and discuss an issue related to the use of virtual memory in Linux kernel.

5.2 Immutable Regions of Linux Kernel

We first describe immutable regions of the Linux kernel that we monitored in the experiment. We define immutable regions as the regions that are critical to the operating system integrity such that any modifications on the regions are deemed malicious. Protecting the integrity of immutable regions should be the highest priority, since modification to immutable regions in the attacker's favor would be the most critical because the immutable kernel region constitutes critical component in the OS and any compromise in this region would seriously affect all the application running on top of the OS. Therefore, our prototype of SnoopMon focuses on monitoring the immutable regions of kernel and it is the main issue in this paper.

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. These should never be modified at runtime. The system call table is another good example of immutable regions. 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.

5.3 Physical Addresses of Immutable Regions

As stated in [20], 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 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 physical addresses of symbol names. The symbol _text and _etext signifies the start and the end of the kernel's text section. The physical address of syscall table and IDT is also determined at compile time so we can make use of them for monitoring.

5.4 SnoopMon

We describe our SnoopMon design and explain how our design meets the requirements we proposed in Section 4. Figure 3 shows key design features of SnoopMon. To prevent any modifications to the immutable region that we explained in Section 5.2, we specifically capture any write operation on those intervals of addresses.

We implemented our SnoopMon as 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 to be separate from the host system. Moreover, the host system cannot access 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.

As in computer systems in general, there are L1 caches that connects the system bus and the processor. So there are two links between the processor and the main memory: L1 caches and the system bus. Modifying L1 cache link in most computers is difficult or sometimes impossible while adjusting the structure of the system bus is relatively more flexible. For instance, ARM architecture can easily adopt Vigilare's system bus architecture.

The Vigilare system has hardware connections to the host system bus to snoop its traffic. However, an attempt to read the main memory content that is in the cache, may not generate corresponding read request on the bus. The bus traffic that indicates read attempts will be generated only when cache-miss occurs. The time that the traffic of write attempts is generated depends on the type of cache: write-through cache or write-back cache. In case of write-through cache, all write attempts by processor may generate the corresponding packets on the bus. In case of write-back cache, the write attempts from 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. Even if an attacker successfully performed the attack, it generates dirty bits in cache lines of immutable 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 snoops. Simply enforcing write-through policy on the host caches would also be an alternative solution.

5.5 SnapMon

To better compare our snoop-based monitoring scheme with traditional snapshot-based method, we also implemented a snapshot-based integrity monitor. Our SnapMon design follows snapshot-based security monitors which were often represented by Copilot [20]. We implemented the hardware of SnapMon by replacing the Snooper with a DMA (Direct Memory Access)-capable memory interface. Hence, SnapMon uses the 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

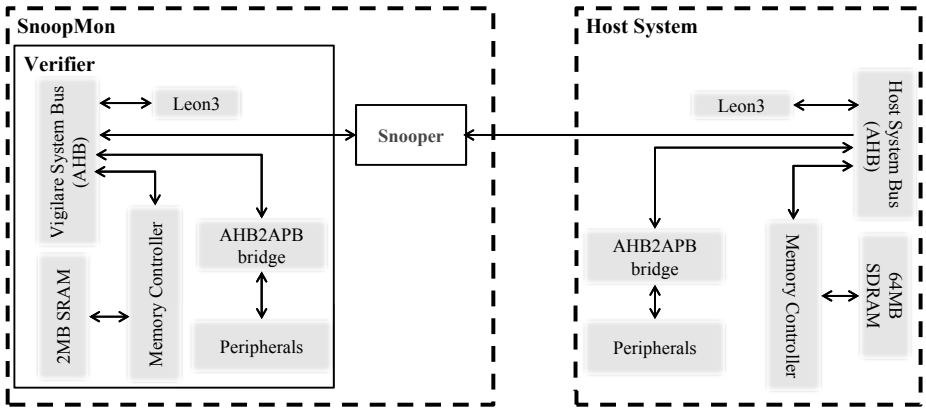

Figure 3: This diagram shows the architecture of our prototype with SnoopMon. SnoopMon has its own memory and peripherals for its independence from host system. It snoops host system bus traffic with Snooper. Snooper delivers traffic that indicates write attempts to immutable regions of host system kernel.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

compares it against the pre-calculated hash value of the unmodified immutable region. The Verifier logic for SnapMon is rather simple; it only compares the two hash values to locate any modifications. In order to implement the monitoring functionality on the hardware, we used 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 was longer than we had expected; it took about 5 seconds to hash them using the processor on Verifier. Consequently, we included a hash accelerator in SnapMon to shorten the processing time to a reasonable level; and it improved to be about 1.3ms to hash one snapshot.

6. EVALUATION

In this section we present results of our experiments about SnoopMon and SnapMon. We performed two experiments to compare the performance degradation and the ability to detect 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 does not have any parameter that possibly affects the result of experiments so we used only one type of SnoopMon.

6.1 Performance Degradation

STREAM benchmark is widely used for measuring the memory bandwidth of a computer system. We used a tuned version of STREAM benchmark [18] 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 1000 experiments to acquire more accurate results. As Figure 4 shows, SnoopMon does not degrade the performance at all, 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 does not include the potential overhead that can be caused by the memory bandwidth consumption on the host with many competing processes for memory access. In our experiment, there is no other process competing for the memory bandwidth except the benchmarks. The performance degradation also depends on the size of the immutable region, as it determines the size of snapshots to be taken and be transferred over. Since the Linux we used is tuned for embedded systems, the size of immutable region of the kernel that we implemented for testing was less than 1 Mbytes.

6.2 Transient Attack

Another objective of experiment was to observe the performance on detecting transient attacks. To compare the ability to detect the attacks, we built a rootkit example that performs transient attack, and measured the probability of detecting a pulse of an attack when we used each monitor.

We implemented a rootkit example for our experiment that meets the definition of transient attack in Section 3. Figure 1 shows how our rootkit example works. It modifies the system call table of the Linux kernel to hook some of the system calls. After t_{active}, it removes its hooks by modifying the system call table as it has been before. After $t_{inactive}$, it hooks the system call as it did before. In our experiment, we measured the probability of detecting one pulse of the attack, depending on t_{active}. We generated 500 pulses and measured how many of them were detected by each monitor.

Figure 5 shows the results of the experiments. SnoopMon detects all the pulses of attacks for all t_{active}, while SnapMon misses many of the transient attacks. 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

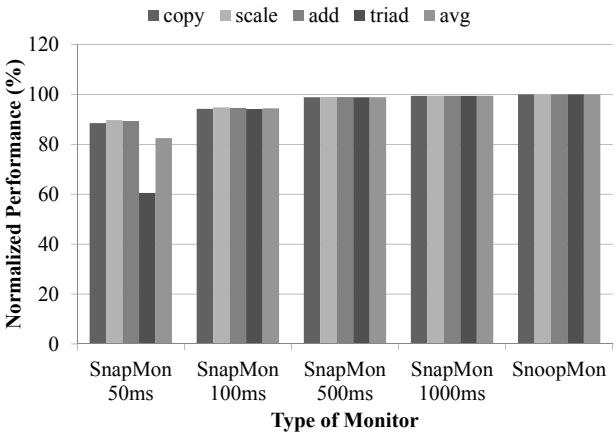

Figure 4: Performance degradation due to each monitor. Numbers in legend indicates t_{active} of each pulse in the attack in millisecond. SnapMon with shorter snapshot interval (i.e., increased snapshot frequency) degrades performance of the host more. For instance, performance degradation for SnapMon with 50ms snapshot interval is 17.5% on average, 10% in the best case and about 40% in the worst case, while SnoopMon has no performance degradation on the host.

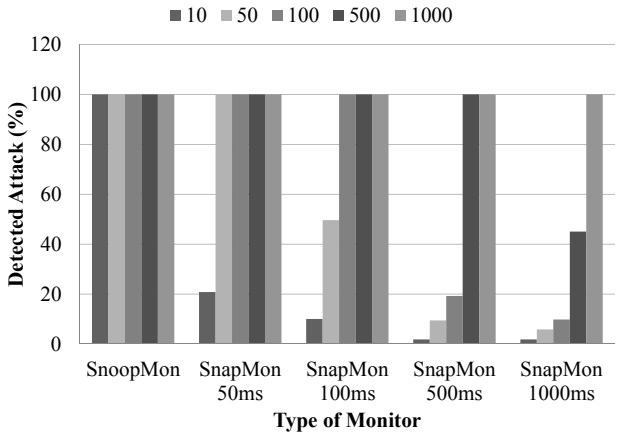

Figure 5: Ratio of detected attacks for each monitor. Numbers in legend indicates t_{active} of each pulse in the attacks in millisecond. This shows SnoopMon detects all the attacks while SnapMon cannot. The graph shows that SnapMon detects all the attacks having t_{active} larger than or equals to its snapshot interval. However, if t_{active} becomes lower than snapshot interval, the ratio of detection decreases as t_{active} decreases. Transient attacks that are active for 1000ms were mostly detected, but when the attack is short-lived t_{active} becomes 10ms, the ratio of detected attack drops significantly ranging from 2% to 20%. If t_{active} become lower than snapshot interval, the ratio of detection decreases as t_{active} decreases.

increase the snapshot frequency significantly for SnapMon to reliably detect transient attacks.

6.3 Discussions

These two experimental results show 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 which uses snoop-based monitoring detects all the transient attacks with little or no performance degradation.

7. LIMITATIONS AND FUTURE WORKS

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

7.1 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 [22].

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 dynamic region.

7.2 Relocation Attack

Our SnoopMon 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 would be capable of detecting 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.

7.3 Power Consumption Analysis

The Vigilare system might increase the total power consumption of the entire system, due to additional hardware. However, the opposite is often true as well because replacing the functionalities of software with a hardware module decreases overall power consumption as it improves efficiency. More work is needed to compare the amount of increased power consumption for each type of monitors.

7.4 Vigilare on Various Platforms

We evaluated the concept of Vigilare on a small and simple embedded system but the concept might be applicable on other platforms: desktops, servers, or application processors for mobile devices. All these platforms have link between processors and memories. Applying Vigilare to these platforms, however, may not be straightforward since each platform has its own characteristics and restriction. Designing Vigilare for each platform would be a part of our future work.

8. 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 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. [9]

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.

8.1 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 possess 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 [14] 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 [22, 23].

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 [1, 2, 3, 4]. 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 [13, 17, 25, 26]. 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. SecVisor, the most well-known work of among such approaches, introduced higher performance degradation than the popular hypervisor software Xen.

Recently, Rhee et al [24] 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.

8.2 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. [16]. Later, X.Zhang et al. proposed IDS (Intrusion Detection System) based on a coprocessor independent from the main processor [30]. Petroni et al. designed and implemented Copilot [20], 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. [8, 21].

Intel also contributed to the trend, by presenting a hardware-based support snapshot-based rootkit detection called as DeepWatch [10]. J. Wang et al. designed HyperCheck [28] 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 [7] 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 [20, 28] 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 [7] 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-based monitor along with Copilot [20] and HyperCheck [28], 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.

8.3 Snooping Bus Traffic

Snooping bus traffic is well known concept as shown in these two prior works. Clarke et al. [11] 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 [19] 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.

9. CONCLUSIONS

In this paper, we proposed snoop-based monitoring, a novel scheme for monitoring the integrity of kernel. 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's SnoopMon 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-powered Vigilare is 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, we also investigated the performance impact on the host system using STREAM benchmark [18], and showed that Vigilare, due to its independent hardware module for bus snooping, imposes no performance degradation on the host. 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.

10. ACKNOWLEDGMENTS

This work was partly supported by VigilSystem, Korea Science and Engineering Foundation (KOSEF) NRL Program grant (No. 0421-2012-0047), the Attached Institute of ETRI, the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2012-0000470), and the Center for Integrated Smart Sensors funded by the Ministry of Education, Science and Technology as Global Frontier Project (CISS-0543-20110012).

11. REFERENCES

[1] Vmware : Vulnerability statistics. http://www.cvedetails.com/vendor/252/Vmware.html.

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

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

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

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

[6] ARM Limited. *AMBA™ Specification*, May 1999.

[7] 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, New York, NY, USA, 2010. ACM.

[8] 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*, pages 77 –86, dec. 2008.

[9] 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*, HotMobile '10, pages 49–54, New York, NY, USA, 2010. ACM.

[10] Y. Bulygin and D. Samyde. Chipset based approach to detect virtualization malware a.k.a. deepwatch. In *BlackHat USA*, 2008.

[11] 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. Technical report, MIT LCS memo-470, http://csg.csail.mit.edu/pubs/memos/Memo-470/smpMemoryMemo.pdf, 2004.

[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] 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*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM.

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

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

[16] D. Hollingworth and T. Redmond. Enhancing operating system resistance to information warfare. In *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, volume 2, pages 1037 –1041 vol.2, 2000.

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

[18] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer*

Architecture (TCCA) Newsletter, pages 19–25, Dec. 1995.

[19] 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*, RTSS '08, pages 481–491, Washington, DC, USA, 2008. IEEE Computer Society.

[20] 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*, SSYM'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[21] 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*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[22] 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*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM.

[23] 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*, ASIACCS '07, pages 194–202, New York, NY, USA, 2007. ACM.

[24] 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*, pages 74 –81, march 2009.

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

[26] 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*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[27] SPARC International Inc. *The SPARC Architecture Manual*, 1992.

[28] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In S. Jha, R. Sommer, and C. Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 158–177. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15512-3-9.

[29] 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*, pages 97 –107, dec. 2008.

[30] 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*, EW 10, pages 239–242, New York, NY, USA, 2002. ACM.