Title: A dynamic per-context verification of kernel address integrity from external monitors

Author: Hojoon Lee, Minsu Kim, Yunheung Paek, Brent Byunghoon Kang

PII: S0167-4048(18)30142-1
DOI: https://doi.org/10.1016/j.cose.2018.02.013
Reference: COSE 1298

To appear in: *Computers & Security*

Received date: 26-6-2017
Revised date: 29-12-2017
Accepted date: 20-2-2018

A Dynamic Per-context Verification of Kernel Address Integrity from External Monitors

Hojoon Lee[a], Minsu Kim[a], Yunheung Paek[b], Brent Byunghoon Kang[a,]*

[a]*Graduate School of Information, School of Computing, KAIST, 291 Daehak-ro Yuseong-gu, Daejeon 34141, Republic of Korea*

[b]*Department of Electrical and Computer Engineering, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 151-744, Republic of Korea*

Abstract

The introduction of *Address Translation Redirection Attack (ATRA)* have revealed a critical weakness in all existing hardware-based *external* kernel integrity monitors. The attack redefines system's memory mappings in favor of the attacker so that the monitored kernel regions are relocated out of the monitor's sight. We provide a generalized approach and a prototype evaluation to prove our proposed scheme for ensuring the integrity of kernel address mapping from external monitors.

With a slight modification on the hardware-side on the host, we were able to enable the monitor to continuously trace *Page Table Base Register (PTBR)* of the host – which is an essential capability in monitoring the host memory mapping integrity.

In conjunction with this hardware feature, we incorporate our findings on the invariant of the kernel memory mapping patterns to implement a dynamic per-context page table monitoring scheme. Our experiment proves the viability of our work in terms of its effectiveness against memory mapping manipulation attacks such as ATRA and satisfies the time constraints required by the proposed per-context mapping verification scheme.

Keywords: External kernel integrity monitor, Address translation redirection attack, Memory mapping integrity, Kernel Security

1. Introduction

*Corresponding author

Email address: hojoon.lee@kaist.ac.kr (Hojoon Lee), pshskms@kaist.ac.kr (Minsu Kim), ypaek@snu.ac.kr (Yunheung Paek), brentkang@kaist.ac.kr (Brent Byunghoon Kang) (Brent Byunghoon Kang)

Detecting kernel-level rootkits is a formidable task, as they make use of the kernel privileges to perform their malicious activities, which can have a system-wide impact. These rootkits can make subtle changes to the OS kernel behavior. These slight changes in the kernel, however, impose a formidable threat to the system integrity. User-level applications depend on the kernel for the utilization of the system resources; operations such as memory management, networking, and so forth, user applications solely depend on the kernel. Another complication in rootkit detection is the necessity of the means of kernel-independent introspection. Conventional in-system introspection software inevitably depends on the kernel, and therefore is unreliable. Even if such software deploys a kernel-level component that acquires the same level of privileges as kernel – and the rootkit – the situation simply turns into an arms race. For this reason, researchers have strived to establish a trusted execution environment either with a protected execution mode within the system [1, 2, 3, 4, 5, 6] or outside the host system [7, 8, 9, 10].

The researchers who proposed *external* hardware-based kernel integrity monitors have claimed better isolation as the main advantages of their approaches. They questioned the security of the hypervisor-based kernel integrity monitors, pointing out that hypervisors are just another layer of software (assisted by processor feature, i.e., Intel `VT-x`) and there have been reports on their vulnerabilities [11, 12, 13, 14, 15]. By monitoring the host system kernel from a completely isolated independent processor, the security of the monitoring software can be protected.

The hardware-based monitors [7, 8, 9, 10] seemed to be a viable alternative until the authors of [16] showed the inherent weakness of the monitors. The attack illustrated by the authors, dubbed *Address Translation Redirection Attack (ATRA)*, redefines the virtual to physical address mappings in the kernel's paging subsystem. By modifying the corresponding physical address destination of the virtual address of the monitored page, an attacker can relocate the page out of the monitor's sight. The hardware monitors are theoretically incapable of detecting such an attack due to their lack of ability to monitor the *Page Table Base Register (PTBR)* of the host. Since each context has its own value of *Page Global Directory (`PGD`)*, it is impossible to reliably trace the currently active *paging structures* without knowing the value in the PTBR. The implication of the presented attack implementation is grave - the hardware-based kernel monitoring approach may become obsolete.

In this paper, we present our approach for a host memory mapping integrity verification that can run from external hardware-based monitors. Our *per-context dynamic mapping verification* scheme enables detection of memory mapping manipulation attacks such as ATRA. In the hope that our work can serve as a reference for designing an ATRA-resilient future external monitor, we explain hardware design amendments and kernel memory mapping invariants that are indispensable for enforcing the mapping integrity for each process context. We implemented a prototype external hardware-based monitor that is equipped with our verification scheme.

The key difference between our design and those of the existing monitors is the *PTBR tracing* feature, where the monitor is notified for context switch events in the host so that the memory mapping integrity can be verified for each context. Based on this feature, we further examined the Linux kernel to find the kernel memory mapping invariants, which can enable us to differentiate ATRA from benign kernel memory mappings. We also define the time constraints that are imposed upon our verification scheme – the rapidly occurring context switch in the host requires our scheme to be finished within a narrow time window, and show that our optimizations overcome the constraints through our experiments.

The remainder of this paper is organized as follows: Section 2 presents an attack model and assumptions. Section 3 provides some background information regarding external monitor, ATRA, and kernel memory layout. Section 4 discusses investigation on the design for an external monitor that can verify the host memory mapping integrity. In Section 5, we present our memory mapping verification design to prevent ATRA like attacks. Section 6 presents our prototype and results of our experiments. Related work is in Section 7 and we conclude in Section 8.

2. Attack Model and Assumptions

In this section, we describe the capabilities of the attacker and the external hardware-based monitor. Also, we describe the assumptions that we make in this paper including the general system configurations.

2.1. Attack Model

We assume that the attacker attempts to subvert the host kernel by modifying critical parts of kernel such as kernel code or important data structures, then launches ATRA to hide the kernel modified memory contents. ATRA launched by the attacker can be *Memory-bound ATRA* or *Register-bound ATRA* which are the two types of ATRA explained in [16].

We also assume the monitor with our ATRA-defense mechanism to be inspecting the host continuously. In regards to the capability of the monitor, we assume it is capable of all previously proposed external monitoring techniques such as memory snapshotting or memory event snooping in order to represent a state-of-art external hardware monitor. The host system runs a commodity operating system which is Linux kernel in our case.

2.2. Assumptions

2.2.1. Secure Boot

We assume the use of a secure boot mechanism; we regard the system is clean until the kernel finishes its startup procedures before allowing user applications to run. This way, the host notifies our monitor of the basic system configurations such as the type of paging mode that will be used during runtime (i.e., `x86-PAE`, `x86-x64`). Also, we expect that the host kernel passes the addresses of the kernel symbols such as the kernel modules that are loaded during boot.

2.2.2. Hardware Modification

We consider the previous work [7, 8, 9, 10] on hardware-based monitors a design for *System-on-Chip (SoC)* and/or possible design suggestions for future commodity architectures. Hence, we assume a certain degree of flexibility in hardware modification. While we regard minor changes to the hardware architecture acceptable, however, we restrain from hardware changes as much as possible for practicality and simplicity.

3. Background

In this section, we provide a background on external monitors, ATRA and kernel memory layout. Specifically, we would like to describe the capabilities and monitoring techniques of the proposed external hardware-based kernel integrity monitors and how ATRA have exploited the monitors. (We will refer to these hardware-based monitors as simply *monitors* or *external monitors* from now on).

Also, we explain the paging on the Linux kernel in detail for the better understanding of our work.

3.1. External Monitor Vulnerability and ATRA

The hardware-based external monitors collect low-level traces of the host system to discover possible kernel compromise. The means of host trace collections proposed in previous work are memory snapshots [7, 9] using DMA, bus traffic snoopings [8, 9], and monitor logic that is integrated into programmable DRAM [10]. The common characteristic of the monitoring mechanisms is that the collected information is from the *physical* address space of the host. Unfortunately, the previously proposed monitors were not designed with the proper consideration of possible attacks on host memory mapping integrity.

ATRA exploited the naive assumption made by the designers of external monitors, to provide rootkits with an ability to completely bypass the monitoring. The monitors assume that virtual to physical address mappings of the monitored regions are either fixed [7, 8, 9, 10], or the monitor relies on host page tables to extract the mappings. ATRA redefines the virtual to physical mappings of the host; the attack assigns new physical addresses to the targeted virtual addresses to redirect memory accesses to her maliciously crafted memory contents.

Authors of [16] proposed two types of ATRA to render the external monitor ineffective. *Memory-bound ATRA* modifies one of the shared paging structures to redirect an address translation to a page frame that is other than the original. *Register-bound ATRA* is an even more powerful form that overwrites the *Page Table Base Register (PTBR)* with the address of a maliciously crafted `PGD`. The attack exploits the fact that none of the existing external monitors are capable of monitoring the register value changes on the host CPU and is shown to be undetectable by the monitor that runs on its own CPU external to the host.

3.2. *Paging and Kernel Memory Layout*

In this paper, we propose design amendments to existing external monitor for ensuring host kernel memory mapping integrity. We briefly explain certain aspects of paging that could aid in understanding the coming sections.

3.2.1. *Components of Paging*

Modern architectures include a PTBR (i.e. `CR3` in x86, `TTBR` in ARM, `CTPR and CR` in SPARC[1].) to store the starting *physical address* of the first entry in `PGD` of the currently running

[1]SPARC uses *Context Table Pointer Register (`CTPR`)* to store the address of the global context table and *Context Register (CR)* to store context numbers which are assigned to each process. We discuss the SPARC MMU architecture in Section 7

process. In general, each process has a unique user address space and thus there is a unique instance of PGD for each process[2].

The Linux kernel adopts a four-level paging model across all architectures. The arrays that implement the four-level translation are called *Page Global Directory (PGD)*, *Page Upper Directory (PUD)*, *Page Mid-level Directory (PMD)*, and *Page Table Entry (PTE)*.

Depending on the architecture and configurations, the number of levels might be less than four. In such cases, the intermediate paging structures might be allocated with a single entry to be essentially ignored [17, 18]. These structures store the physical addresses of the next-level. We will be using these four types of Linux data structure names to denote the arrays that are used in address translation. Also, we will be referring to these structures commonly as *paging structures* throughout this paper.

One fact that needs to be stated before going into the later sections is that conceptually there is only one kernel address space across all processes within a system. Thus, while a PGD is unique to each process to support per-process user address space, PUD, PMD, and PTE that correspond to the kernel address range are shared among all processes. For this reason, a PTBR-manipulating attack must be launched in a per-context manner. For instance, one can hook a function that is always executed when a process is scheduled as illustrated in [16]. Conversely, an attack on the shared intermediate and leaf paging structures (PUD, PMD, PTE) affects the kernel-mode execution of all processes.

3.3. External-Monitoring Techniques

Before we explain our proposed design amendments, it is necessary to clarify the extent of the monitor capabilities. As discussed, there are a few monitoring techniques employed by external monitors. We see that these techniques provide mainly two types of input to the monitors. The first is *events* that are generated asynchronously when a pre-defined range of monitored memory region receives memory writes [8, 9, 10]. The technique has been implemented with either using a bus-snooping hardware [8, 9], or by leveraging the FB-DIMM technology [10]. The second type is taking *memory snapshots* on a regular or random interval,

[2]in ARM architecture there are two PTBRs (TTBR0 and TTBR1), and they can be concurrently used such that one maps kernel space and the other maps user space. However, Linux uses only one PTBR for per-process PGDs

where the external monitor makes synchronous inquiries for the current memory content via DMA [7, 8, 9].

Detecting anomalies or malicious modifications to the kernel using such low-level inputs requires *invariants* [19, 20, 21, 22, 9], or a set of relations among registers and memory locations that can ensure the integrity of a certain kernel behavior. Existing works enforce pre-defined invariants to discern the rootkit presence in the kernel. For instance, kernel code and data protection scheme in existing work [7, 8, 9, 10] utilizes a simple yet clear invariant where any write attempt or a modification to the static region such as the system call table is deemed as malicious rootkits behavior.

A bit more complex invariants are the ones that involve events. These event-based invariants define one or more conditions *C* for an event *E*. The use of these invariants are abundant in hypervisor-based work [23, 5, 24]; generally, a VM-to-hypervisor trap (i.e., *VMEXIT*) is used as an event, and the register values and memory contents are tested as the condition for a certain malicious activity in the VM. A recently proposed hardware-based monitor also deployed a similar technique using *bus-snooping* [9], which was used for monitoring kernel module loading that occurs during runtime.

4. Challenges in Verifying Mapping Integrity From External Monitors

4.1. Identifying Malicious Mappings

The more important issue in the host address mapping verification is how we can differentiate attacks from normal mapping patterns. In other words, it is necessary that we investigate the kernel mappings to see if there are any useful invariants that can be extracted. As we will explain, different sections of kernel address space have varying mapping characteristics. For detection of attacks, it is necessary that invariants for these characteristics must be studied.

In addition, we refrain from host kernel modification. It may be possible to optimize kernel address mappings for external monitoring by modifying the memory allocator of the kernel. For instance, we could fix the address translations for all regions of kernel address space as well as the drivers that may be loaded during runtime. However, this not only impairs the practicality of our monitor but also undermines the efficiency and design principle of kernel memory management implementation. If a modification is to be made to the allocator, it has to be devised with a comprehensive consideration. Thus, we present a detailed investigation on address mapping invariants that are available in the current unmodified kernel such as Linux and their enforcement in this paper.

4.2. Locating Paging Structures

The PTBR is the only trustworthy and kernel independent pointer to the currently active page tables from the monitor's viewpoint. Hence, the ability to detect the event when the PTBR is loaded with a new value is indispensable in ensuring the integrity of address mapping translation. Without this capability, there is no reliable way to discern which blocks in kernel memory are PGDs that represent the address spaces of processes.

Even when the monitor possesses the ability to trace PTBR, complications remain. First, it can be burdensome for the monitor to observe all PGDs that are present in the system. PGD will be created for each process dynamically at runtime. The monitor may deem all unseen addresses that first appear in PTBR as the PGD address for a new process, then start monitoring the region to make sure the entries in PGD not modified. The PGD needs to be continuously monitored even when the corresponding process is scheduled out, because the process can be scheduled back later with the same PGD that was used before. If the attacker is able to modify the PGD while the corresponding process is scheduled out, the maliciously modified PGD will be used when the process runs again. In such case, the regions that need to be monitored would pile up. When the host has N processes present and the size of a `PGD` is M bytes, the monitor must be constantly monitoring $N * M$ bytes of memory. This additional inspection should be performed on top of other integrity monitoring tasks. Second, it is a non-trivial task to constantly check the validity of the monitored `PGD`s. The PGDs could have been deallocated or already reallocated to be reused for another process. This indicates that the external monitor must be constantly tracking process states to find which PGDs have become defunct as the process had exited.

As we will explain in Section 6, our design decision for the issue is to perform just-in-time PGD verification for each address space change.

4.3. Timely Monitoring

The just-in-time PGD verification in our design performs checks in-between address space changes. This means that we begin a verification as soon as a process is scheduled in to load the PTBR with its PGD pointer. This had initially introduced a timing issue where another address space change can occur while a verification of PGD is still in progress.

The PTBR is constantly updated as a *process context switch* occurs in the system; this register always contains the pointer to the PGD of the currently running process. External monitors that run in parallel to the host inevitably face timing issues; for example, PTBR for next process may change to signify a new address space while the PGD verification routine for the current process has not yet finished. We elaborate on this issue in Section 6.4, and present our experiments relating to the issue in Section 7.

5. Defining Mapping Invariants for Kernel Address Space

In this section, we explain our investigation of possible invariants for kernel address mappings. More specifically, we describe how the mapping characteristics differ for the sections in the kernel.

5.1. Tracking Page Table Base Register

As stated in the previous section, the ability to track PTBR is an indispensable requirement for ensuring the integrity of kernel address translation mapping from external monitors. We meet this requirement by making a minor change in the host CPU. More specifically, we modify the MMU (Memory Management Unit) hardware on the host CPU to trigger an interrupt to our external monitor for all PTBR change event. In addition, we present our dynamic per-context monitoring of paging structure, utilizing this essential hardware support in Section 6.

5.2. Kernel Memory Mapping Invariants

The amended hardware lays a foundation for triggering the monitor to check the host memory mapping integrity for each PTBR change on the host. Based on this known property, we need some form of *invariant*s that could be used to determine if a virtual-to-physical address mapping is benign or malicious. The PTBR tracing capability implemented on our monitor hardware generates a notification *event* for each PTBR change as they occur on the monitored host.

The Linux kernel as well as other commodity OS kernels have mapping patterns that can be induced into invariants. The kernel memory mappings can be either *linear* and *non-linear* depending on the memory regions. We use the term linear mapping here to refer to a mapping $M : V \rightarrow P$ that can be defined by an offset X where $V - X = P$. Whereas non-linear mapping refers a mapping that is not necessarily linear and often arbitrarily changes during runtime.

Another important characteristic of kernel mappings is whether the mappings are formed at the system boot time and remain *persistent* or *dynamically changing* throughout the runtime. We will explain the locations and characteristics of the regions within the kernel, and our definition of their invariants in the rest of this section. Note that our scheme is not architecture-dependent, since these mappings are determined by the operating system.

In the rest of this section, for the simplicity of the description, we will assume the default settings of `x86-PAE` Linux when explaining kernel memory regions due to the general familiarity with the specific architecture.

5.2.1. Linear and Persistent Memory Mappings

The linearity and persistence of the kernel memory regions are illustrated in Figure 2. The *Physical Mappings*, which constitutes about 88.7% (887MB / 1GB), the majority of kernel memory space, both linear and persistent. This region ranges from a symbol called `PAGE_OFFSET`, the boundary between user and kernel memory space, to a symbol called `high_mem`. The `PAGE_OFFSET` is normally `0xc0000000` in most architectures including `x86-PAE`. The value of `high_mem` may vary depending on the size of physical RAM, but it was `0xf77fe000` on our test machine, the specs of which can be found in Section 7.

The Physical Mappings region contains the initial kernel image including kernel code as well as kernel objects that are dynamically allocated through the *Slab allocator*[17] of the Linux kernel. The Slab allocator is an abstraction built on top of the elementary memory management implemented by paging. The slab allocator manages a large portion of the Physical Mappings region to support kernel memory allocation functions such as `kmalloc()`. In this way, an efficient allocation algorithm can be implemented and the necessity for page table modification for each memory allocation is removed.

The invariant for the linear region is that the linearity of the region must be kept intact since the boot time (i.e., $V_{addr} - P_{addr} = OFFSET_{v \rightarrow p}$). With the known constant offset, the invariant can be used for a quite powerful defense method against ATRA, because the external monitor can perform host address translation in the same way as the MMU on the host processor. Since the physical address in PTBR can be traced with our hardware monitor, ATRA-like attacks cannot bypass our scheme.

The constant offset is determined during compile time. In turn, the value can be effortlessly extracted from the kernel image. Also available in the Linux kernel image is the base address to which the image is loaded during boot time. Hence, these two compile-time constants define the invariant for the linear and persistent kernel mappings.

5.2.2. Non-Linear/Persistent Memory Mappings

A region in kernel memory space called *fixmaps* is quite peculiar in that it is mapped at compile time and meant to stay immutable during runtime but not necessarily linear [18, 25].

Unlike `physical mappings` region, there is no legitimate $V_{addr} \rightarrow P_{addr}$ mapping that can be calculated with an offset. It is imperative that the compile-time generated mappings of the `fixmaps` region should be obtained from the compiled kernel binary and the verification of secure boot should be deployed to ensure that the mappings are correctly applied during boot time. With a guarantee that these mappings are safely initialized, the runtime invariant is rather straightforward. These mappings should not be modified at runtime along with the mappings for the `physical mappings` regions.

5.2.3. Mapping Invariants in Non-linear/Non-Persistent Regions

The *vmalloc region* and *persistent mapping* regions are neither linear nor persistent. The Linux kernel provides a kernel function called *`vmalloc()`* to allow allocation of physically non-contiguous memory in the `vmalloc` region [18]. The purpose of the region is to map physical page frames, additional to the 896MB `physical mapping` region, into the `high_mem` region. However, its use is generally discouraged except for some kernel drivers [26].[3]

The `persistent mappings` region is used for mapping `high_mem` pages into kernel address space using `kmap()` function. The name may be misleading in the context of this paper; mappings made to these regions are in fact not persistent in our definition of persistent mapping. The region maps for `high_mem` pages that may be used a bit longer than the ones mapped by `vmalloc()` which are mainly for temporary and large buffers. It is generally recommended that the pages mapped by `kmap()` be freed by `kunmap()` after a short-term use [18, 27].

The two regions are seldom used for the data structures of critical kernel subsystems such as the scheduler, memory management, and file system. However, the kernel modules, which are

[3]When someone suggested the use of `vmalloc()` for his code in a kernel developer mailing list, Linus Torvalds himself replied "... vmalloc() is NOT SOMETHING YOU SHOULD EVER USE! ... There's a few valid legacy users (I think the file descriptor array), and there are some drivers that use it (which is ***, drivers are drivers), and it's _really_ valid only for modules. Nothing else."

generally allocated in `vmalloc` region, are substantial parts of the kernel. While these regions are non-linear and also non-persistent such that an invariant cannot be defined, our investigation has revealed that two conditions in which an invariant can be defined for some subsets of the region, which include the memory that was allocated for the module's text region.

First, although the mappings are not meant to be persistent, there are cases where they stay persistent. For instance, essential drivers – say a keyboard or graphics driver – are loaded in the `vmalloc` region during the system boot time, it is unlikely that these are unloaded before the system shutdown. Therefore, we can retrieve their locations from the host during its boot time when the mappings are determined and can monitor them as if they were persistent mappings. Likewise, as with the invariants for non-linear and persistent mappings, a secure boot mechanism must be deployed to safely record and transfer these boot-time determined addresses to the external monitor.

The second condition is when we could capture mapping creation/destruction event by tracking the memory events. The kernel module list such as LKM (Loadable Kernel Module) in Linux can be an example of this type of invariant. By keeping an eye on the kernel module linked list, the monitor can capture the address in `vmalloc` region to which the module is being loaded. Also, it should be noted that the kernel module linked list – the data structure that is used for the bookkeeping the list of LKM, not the LKM object itself – is within the `physical mapping` region, such that its address integrity can be assured by simply ensuring the mapping invariant for the kernel linear region.

We see that monitoring the non-linear and non-persistent kernel regions that are allocated by `vmalloc()` is a daunting task. However, the use of vmalloc is seldom used for important kernel objects; its use is largely limited to the kernel drivers. A recent work by Rhee et. al. [28] also reported that all the kernel memory allocation is indeed mostly made in the linear regions, which was shown by capturing all kernel memory allocations.

6. Address Mapping Verification Design

We have explained two main building blocks – the PTBR tracing capability and kernel mapping invariants – that are crucial in enabling host memory mapping verification from an external monitor. In this section, we describe how the building blocks are molded into our dynamic per-context mapping verification scheme.

Before we describe our dynamic per-context mapping verification scheme, we explain the capabilities of our prototype monitor. Our monitor is capable of detecting host memory modification like the existing external monitors [8, 9, 10].

The PTBR tracing implemented in our prototype hardware plays a central role in our mapping verification scheme. The lack of this feature in the existing work was the Achilles' heel, the critical weakness exploited by ATRA. Figure 3 illustrates our modifications to the host MMU which is an open-source SPARC architecture processor. We duplicate two registers that reside in the *MMU Register File*: *Context Register (CR)* and *Context Table Pointer Register (*`CTPR`*)*. The duplicated registers are fed into the external monitor to notify the context switch that occurs in the host. Note that these duplicate registers are just mirrors of their original counterparts; the data being written into the originals are simply copied onto the duplicated registers.

Our verification design implements enforcement of the kernel mapping invariants that we explained in the previous section. In essence, the design translates the abstract invariants into a set of verification routines that run in the external monitor.

Accessing a memory page that has a mapping invariant $M_{invariant} : V \rightarrow P$, requires the MMU to access a sequence of {`PGD`, `PUD`, `PMD`, `PTE`} in respective order. Our goal is to verify that these paging structure sequence still retain the correct values as they are put into use.

Note that each process is given a unique `PGD` during its creation, while the rest of the paging structures are shared across all processes for kernel memory range. This means that we have a moving target to monitor. We need to be monitoring the `PGD` of the currently running process among as many `PGD`s as the number of processes in the system.

6.1. Verifying Process Context Switches

In our scheme, the first-level structure `PGD` is verified for each context switch event in the host. When the host PTBR is modified, the monitor is notified of the memory address value stored in PTBR. In accordance, the monitor performs two actions.

First, the monitor dynamically configures the memory snooper to update the starting physical address of the `PGD`. From this point on, any modifications made to `PGD` is observable by our monitor. The second action performed by the monitor is to hash-compare the entries of the loaded `PGD`. The hash comparison here is to ensure the `PGD` is not initialized by the adversary with malicious values or has been modified during its inactive period. This is necessary because our monitor is only aware of the currently active `PGD`.

Note that we monitor only the currently-in-use `PGD`s by capturing context-switching events, oblivious of the inactive `PGD`s. We concluded that this is a simple yet robust way of monitoring the host `PGDs`. In fact, it is not viable nor effective to monitor inactive `PGD`s. We cannot assume anything about the state of the `PGD`s; They could have been deallocated or already reallocated to be reused for another process. Furthermore, the adversary may create a new `PGD` to launch an attack as shown in [16]. Since the adversary can use an arbitrary memory chunk as a `PGD` by supplying its pointer to PTBR, monitoring only the `PGD`s that are declared as such by the kernel semantics is not enough. As such we conclude that a seemingly simplistic stateless monitoring scheme that reacts to each PTBR change to verify loaded `PGD`s, is in fact the most dependable.

This per-context `PGD` monitoring effectively detects the loading of a `PGD` that differs from our hash value. The Register-bound ATRA [16] injects a piece of code that forces the victimized process to load the maliciously crafted `PGD` through IDT hook. However, we verify the mapping values of kernel `PGD` entries as it becomes active (i.e., the PGD that is referenced by the newly updated pointer value in PTBR) and watch this active PGD while it is in use (i.e., until the PTBR value changes for next process to run). This is why preparing a maliciously formed `PGD` or modifying the existing ones do not pose a threat to our scheme, as we would verify it when it actually matters or right at the moment when the PGD with malicious content is being loaded for the attack.

Upon verification of PGD integrity, the monitor will be dynamically configured to watch the memory region of the current PGD. One may wonder why we need to monitor the PGD since we can detect the malicious modification of the PGD next time when this modified PGD is loaded into the PTBR. This would be true if the PGD modification stays unchanged after context switch, however, at the context switch event, the attacker can wait for the PGD hash verification to be finished, and then make the malicious modification of the PGD right after the verification is completed, making successful TOCTOU-style ATRA attack. Moreover, the attacker can also launch an transient ATRA attack by simply restoring the correct value of PGD entries, right before the attacked process is scheduled out.

For some hardware-based external monitor, dynamically configuring the PGD location to be monitored for each PTBR value may incur slight overhead for setting up the hardware register of the monitored memory location, or for dynamically configuring the memory region for each process switch event. If this overhead can be burdensome or if the external monitor is not capable of dynamically setting the monitored region, as a design choice, the PGD check can be applied at a random interval, making the attacker difficult to predict when to modify the PGD entries, although this method would entail a chance of not detecting the transient attack sometime, but it will not require the setting PGD memory locations for every context switch. However, our prototype has chosen the method of making the hash checking of PGD be done at the start and the content of PGD be monitored while in use, making transient and TOCTOU attack infeasible.

6.2. Monitoring Shared Paging Structures

Note that the kernel mappings in the remaining paging structures (`PUD`, `PMD`, and `PTE`), that stem from a `PGD` are shared across all processes as explained in Section 3.2. This property simplifies the monitoring; we can calculate the addresses and the hashes of the entries that need to stay unmodified and then we can monitor them during runtime with a hash check on the paging structures captured by snapshots or a memory snooper module.

Since we assure that the kernel space entries of the currently active `PGD` have known-good values such that they point to the shared kernel paging structures, we can simply monitor these kernel-global data as immutable.

6.3. Tolerating Memory Status Flags

It is necessary to consider various architecture-dependent flags in the entries of the paging structures when monitoring them. Flags such as *dirty*, *accessed*, and permission flags may be legitimately modified by kernel memory management. This implies that if the flags have been updated, the hash value of the paging structures will not match even though the mapping stays the same since the hash is taken over the entries that include the flag bits.

To handle this issue, in our prototype design, we modified our hardware-based snooper module to create holes in the memory snooper's monitored address settings so that the modifications to the flags are ignored. There may be a few flag toleration methods that are specific to varying monitor implementations. The first method would be to create holes in the watched region as we did for our prototype. The second method would be to generate a list of allowed values for each flag combinations. An example of such value would be a combination of the legitimate mapping value and read-only permission flag.

6.4. Time Constraint on Verification

We found that the rate of context switches in the host processes poses a formidable challenge to our mapping integrity verification scheme. Our memory mapping verification design inspects per-context paging structure for each context switch event. Context switches are one of the most frequently occurring events in a system; thus we should carefully consider the time constraints, where the verification time should not take longer than any interval for two consecutive context switching events. Otherwise, it would not be able to detect an ATRA targeted for a process that runs shorter than the verification time.

In our design, we concluded that our mapping verification performed for each PTBR change must not take longer than the minimum time slice that can be assigned to a process. This is to prevent verification tasks from piling up on the external monitor queue when a large number of processes are running on the host incurring frequent context switches.

As discussed, our scheme involves hashing of the per-context paging structure, the PGD. In our preliminary testing, we found that the cycle-hungry *SHA-1* hash function burdened our scheme in meeting the time constraint requirement. To alleviate this issue, we have implemented a generic SHA-1 hash calculator hardware. With this hardware module, we were able to reduce the execution time of the verification routine well within the time constraints. The related experiment results will be presented in Section 7.

If the attacker could somehow force a process to spend less time on CPU than the minimum time slice, the monitor may not be able to promptly handle the verification of the next context switch. However, exploiting this property to evade our check is infeasible because such attack would require completely changing how scheduler operates, and also rearranging rapidly changing process queue. Thus, we see that it is rather difficult to achieve a system-wide mapping manipulation in such way.

7. Prototype Evaluation

We implemented a prototype of our dynamic per-context mapping verification scheme on an FPGA board. We used a SPARC-architecture LEON3 processor on both the host and the external monitor. (We chose SPARC since the x86 CPU architecture owned by Intel is not available for prototyping. However, we believe the minimal hardware modification required for notifying the PTBR value to an external monitor can be readily implemented by the vendor.) Both the monitor and host run at 50 Mhz and equipped with 64MB SDRAM. For the operating system kernel on the host, we used Snapgear Linux (2.6.21.1 kernel version) that is patched for the particular processor architecture [29]. On the other hand, the external hardware monitor in our prototype runs a bare metal kernel integrity monitoring software without an OS. - In addition to snooping and snapshotting techniques, as discussed in Section 6, we implemented the PTBR tracing hardware feature onto the LEON3 processor.

With this prototype, we ran experiments with the following research questions. First, we would like to test the validity of verification approach based on kernel linear region invariants. Second, we show the feasibility of the implementation where the time for dynamic per-context mapping verification is shorter than the minimal context-switch interval, the minimum time slice used by the scheduler in the system. The Linux version used in our prototype has a default value for the minimum time slice as 750 μs (i.e., 0.75 ms). We also confirmed that a recent version of Linux (such as 3.13.0) also has the same value.

7.1. Monitoring Shared Paging Structures

We applied our verification scheme to our host system and tested its validity. As we explained in section 6, we monitor all shared paging structures (i.e., PUDs, PMDs, PTEs in Linux terminology) that correspond to kernel address space. By examaning the compiled kernel image, we were able to collect the exact addresses of the different sections in our kernel.

7.1.1. Kernel Memory Layout

In our host kernel, the break between user and kernel was at `0xf0000000`. The memory layout of the kernel is inspected to be as the following:

The SPARC architecture MMU uses two registers to point to the per-context paging structure called *Context Table (CTX Table)*. The `CTPR`, as its name implies, stores a 30-bit base to the global context table. The `CR` stores an index to the global context table called *CTXNR*, of which its entry is uniquely assigned to each process. This unique MMU architecture may seem strange to those who are not familiar with the SPARC architecture. While the `x86` and `ARM` architecture stores physical address to per-process `PGD`. The SPARC `context table` is a globally used array that stores the physical address pointer to all processes. The significance of this difference is that there is an additional per-context paging structure that needs to be monitored in the SPARC architecture. The entry in `context table` should be verified along with the `PGD`.

In this evaluation, we experimented with locating and monitoring the paging structures for the linear and persistent `physical mappings` region and a kernel module that is loaded into a known address during the system boot time (`0xfe602000`). From Figure 5, we can see that the `physical mappings` region is mapped to `0xf0000000` to `0xf4000000`.

7.1.2. Memory-bound ATRA Detection Test

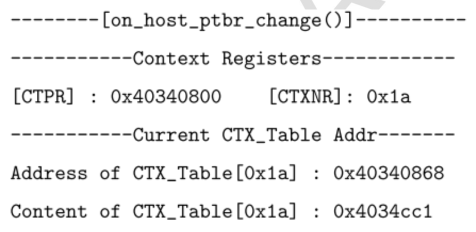
To demonstrate the effectiveness of our prototype against memory-bound ATRA [16] that modifies in-memory paging structure, we developed a kernel module that creates a non-linear mapping in the kernel's physical mappings area. The module walks the page tables for the two large pages that map the phyiscal mappings region: `A`(`0xf2000000`) and `B`(`0xf3000000`). Then, it overwrites the `PMD` of `A` with that of `B`. With the page table modification, `A`'s virtual address (`0xf2000000`) now maps physical address of `0x43000000`. As a result the virtual to physical mapping of `A` is no longer linear.

As explained in section 6, we monitor We configured the snooper module in our monitor prototype to continuously monitor kernel's shared intermediate page table structures. Our prototype successfully detected the attack performed in our ATRA attack kernel module. Since the attack illustrated in our ATRA kernel module modifies an address mapping of the physical mappings area, the modification of the mapping is detected and reported by our prototype. The snooper module on our prototype reported memory modification at `0x40340fc8` (physical address) which belongs to the `pmd` page that map the physical mappings area.

7.1.3. Per-Context Verification

Our verification for the per-process paging structure starts as the monitor observes a new PTBR value. The output capture below illustrates the event seen by the monitor.

Comment [U1]: TYPESETTER: Please insert picture near here as per the manuscript.



```
--------[on_host_ptbr_change()]----------
-----------Context Registers------------
[CTPR] : 0x40340800    [CTXNR]: 0x1a
-----------Current CTX_Table Addr-------
Address of CTX_Table[0x1a] : 0x40340868
Content of CTX_Table[0x1a] : 0x4034cc1
```

The content of the currently indexed CTX Table is the base physical address of the process PGD. We configure our memory snooper to shift the monitored areas with respect to this currently active PGD. Finally, we send the address range of the PGD entries that map the monitored regions to the hash module.

In our particular experiment where we monitor the physical mappings region and a kernel module, a total of 5 PGD were monitored and hashed. (5 for physical mappings region and 1 for the kernel module) This is due to the fact that our host kernel maps the physical mapping region with four 16M pages supported by the SPARC architecture. The remaining one entry is for the PGD that maps the monitored kernel module.

7.1.4. Register-bound ATRA Detection Test

To evaluate the effectiveness of our approach against register-bound ATRA [16], we created a sample register-bound ATRA kernel module and ran our defense scheme against it. The module references the CPTR register to locate the context table and PGD. It then makes copies of the table and PGD then fills the two tables with random values. Finally, the adversary loads CPTR with the physical address of the malicious duplicate context table (system crashes but this achieves ATRA-like behavior). This forces the process to access a rogue PGD in memory translations that has been modified by the adversary. The detection was rather straightforward; our prototype detected the CTPR change and generated the hash of the PGD entries that correspond to kernel's physical mappings region to find that the entries have illegal mappings.

7.2. Meeting the Time Requirement

We performed a set of experiments to measure the total cycles consumed by the mapping integrity verification routine that is executed for each host PTBR change. We define our time constraint as the following inequality and Figure 6 illustrates the factors in this time constraint.

$$T_{adjust} + T_{HashCheck} < T_{MinContext\ Switch}$$

T_Adjust is the time needed for locating the PGD, adjusting monitored memory regions, and initializing the hash module. We measured the time with clock() function provided by the sparc-elf-gcc toolchain. The function returns the elapsed time in microseconds, and we measured T_Adjust to be approximately 2 μs after a hundred trials. The cycles consumed by the hash module was measured by a hardware cycle counter implemented in the FPGA board. Figure 7.2 illustrates the execution time of the entire verification routine with varying number of `PGD Entries`. A few combinations of varying entry sizes and numbers of entries were tested to represent other architectures. The trial with five entries illustrates execution time for the experiment explained in the previous subsection: per-context monitoring of `physical mappings` and a kernel module. Note that the minimum time slice for our prototype host was 750 μs.

As shown through the experiments, our verification scheme implemented on prototype meets the time constraint requirement. Even with the worst case (64bit and 1024 entries) our verification scheme manages to finish within the minimum time slice. This shows that our verification scheme is able to verify each and every PTBR change without skipping one. By meeting this time requirement, we can ensure that this defense scheme is applied to all running processes.

8. Related Work

In this section, we present the related works for our dynamic per-context mapping verification scheme for external hardware monitors to handle memory mapping manipulation attacks. We first list the currently burgeoning efforts in external hardware monitoring schemes, all of which are by design vulnerable to ATRA, and then we discuss some prior works of literature that briefly described some form of memory mapping manipulation attacks. Finally, we mention a few previous methods on inducing and utilizing the kernel invariants.

8.1. External Monitors

There have been a number of approaches to leveraging an external hardware component as their trust anchors for securely monitoring the host kernel. Copilot [7] proposed a PCI card as an external component that can perform periodic hash checks on the kernel static regions via DMA. Vigilare [8] presented a co-processor attached to the host, which can steadily monitor the integrity of kernel static regions by snooping memory bus traffic. It was designed to overcome the limitation of the snapshot-based monitoring scheme, which cannot detect transient modification between an interval of snapshots. KI-Mon [9] extended snapshot-based scheme with callback-based verification that can efficiently monitor dynamic kernel data. MGuard [10] presented a snooping-like scheme implemented in programmable DRAM to address the impracticality of the previous snoop-based monitors that require the modifications of the processor internal for extracting memory bus traffic between core and memory controller. Some of the hardware-based works mentioned possible relocation attacks [7, 8]. However, they are not aware of the fact that their monitors were inherently incapable of detecting ATRA.

8.2. Memory Mapping Manipulation Attack

A few researchers [2, 23, 30] have mentioned the possibility of relocations attack, similar to ATRA, which can subvert their suggested defense system. Although the researchers have also suggested the heuristic mitigations against the attack, they could not convince that the attack can be realized. Jang et al. [16] proved that the attack could be realistic by implementing the attack on x86 architecture, and showed that the attack could subvert existing external hardware monitors in detail. The attack could poison page table data structures, not being detected by external monitors, by modifying the values from PTBR that could not be accessed by the monitors. They also described the challenges in mitigating ATRA by showing the ineffectiveness of the possible countermeasures that even seemed to be theoretical. As the paper mentioned, addressing the challenges (i.e., race condition and protection of all the pointers related to the page structures) is a difficult task and also seems to be impossible. Since external monitors fundamentally have no view of the host processor, the challenges in mitigating ATRA become more difficult. Therefore, in this paper, we address this issue by slightly modifying host processor to notify the active PTBR value to the external monitor. Also, we simplify the complexity in protecting mapping invariants associated with page structures.

8.3. Kernel Invariants

Gibraltar [20] suggested a kernel-level rootkit detection technique that can automatically extract the kernel invariants of both control and non-control data. The invariants are generated by using pre-defined invariant template and kernel objects that are extracted from the memory snapshots of an uncompromised kernel. Dolan-Gavitt et al. [21] presented the robust kernel data invariants that are difficult to evade. Robust invariants were extracted from the data fields that could not be modified with random values since the modification cause the kernel crash. OSck [3] presented hypervisor-based kernel integrity monitor by checking the data object with type invariant from kernel source code. SigGraph [22] suggested graph-based structural invariants, which enables brute force scanning of arbitrary kernel objects, instead of value invariants. Liakh et al. [25] explored the memory property of Linux kernel that violates $W \oplus X$ (does not allow both writable and executable memory page). Total-CFI implements punctual guest OS view reconstruction to identify guest kernel semantics to detect occurrences of software exploits [34].

8.4. ATRA and Mitigation on VMI

Virtual Machine Introspection (VMI) leverages the hypervisors to implement guest kernel monitoring tools [3, 5, 23, 31, 4, 28, 32]. Hypervisors in general has access to a full saved state of the paused guest virtual machine during *VMExits*, and thus hypervisors are theoretically capable of introspecting the guest system for possible ATRA. However, the latest memory virtualization that is commonly referred to as *nested paging* (e.g., Intel VT-x's EPT and AMD SVM's NPT) eliminates hypervisor intervention on guest kernel's page table updates for improved performance. SecPod addresses the issue with its paging delegation mechanism and privileged instruction execution trapping [33]. The kernel memory mapping invariants that we discussed in this paper may aid both hardware-based or hypervisor-based kernel introspection tool developers in monitoring the monitored system's page tables.

9. Conclusion

In this paper, we presented a dynamic per-context memory mapping verification scheme that can amend external monitors in its lack of ability to cope with ATRA – the recently proposed attack that completely bypasses external monitors. We also provided a specific hardware design amendment and comprehensive kernel memory mapping invariants that are essential in implementing our scheme. Our mapping verification scheme is built around the new hardware feature called PTBR tracing, providing the basis with which the invariants are enforced. The monitored regions for the invariants are adjusted for each context switch on the host in order to consistently monitor the currently active paging structures. In our experiments, we detailed how these invariants can be practically enforced. We also carefully considered the time requirement for the verification routine; our measurements show that the execution time for our verification routine for each context switch is sufficiently lower than the scheduler's minimum time slice. Thus, we conclude that our illustrated methods can be incorporated into the external hardware-based monitors, making them no longer vulnerable to memory mapping manipulation attacks.

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, School of Computing KAIST). His research interests include trusted execution environment, hardware-based kernel integrity monitors, and virtual machine introspection.

Minsu Kim received the BS degrees in computer science from KAIST, Korea, in 2011. He is currently working toward the PhD degree at the Graduate School of Information Security in KAIST's School of Computing. His research interests include defense measures against code-reuse attacks.

Yunheung Paek received the BS and MS degrees in computer engineering from the Seoul National University, Korea in 1988 and 1990, respectively. He received the PhD degree in computer science from the 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 retargetable compiler. He is a member of the IEEE.

Brent ByungHoon Kang received the BS degree from Seoul National University, the MS degree from the University of Maryland at College Park, and the PhD degree in computer science from the University of California at Berkeley. He is currently an associate professor at the Graduate School of Information Security (GSIS) at Korea Advanced Institute of Science & Technology (KAIST). Before KAIST, he has been with George Mason University as an associate professor in the Volgenau School of Engineering. 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.

References

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

[2] J. Wang, A. Stavrou, A. Ghosh, Hypercheck: A hardware-assisted integrity monitor, in: S. Jha, R. Sommer, C. Kreibich (Eds.), Recent Advances in Intrusion Detection, Vol. 6307 of

Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp. 158–177, 10.1007/978-3-642-15512-3-9. URL http://dx.doi.org/10.1007/978-3-642-15512-3-9

[3] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, E. Witchel, Ensuring operating system kernel integrity with osck, in: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS ’11, ACM, New York, NY, USA, 2011, pp. 279–290. doi:10.1145/1950365.1950398. URL http://doi.acm.org/10.1145/1950365.1950398

[4] R. Riley, X. Jiang, D. Xu, Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing, in: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection, RAID ’08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 1–20. doi:10.1007/978-3-540-87403-4_1. URL http://dx.doi.org/10.1007/978-3-540-87403-4_1

[5] A. Seshadri, M. Luk, N. Qu, 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, ACM, New York, NY, USA, 2007, pp. 335–350. doi:10.1145/1294261.1294294. URL http://doi.acm.org/10.1145/1294261.1294294

[6] Y. Chubachi, T. Shinagawa, K. Kato, Hypervisor-based prevention of persistent rootkits, in: Proc. 2010 ACM Symp. Appl. Comput., ACM Press, New York, New York, USA, 2010, pp. 214–220. doi:10.1145/1774088.1774131. URL http://portal.acm.org/citation.cfm?doid=1774088.1774131

[7] N. L. Petroni, Jr., T. Fraser, J. Molina, 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, USENIX Association, Berkeley, CA, USA, 2004, pp. 13–13. URL http://dl.acm.org/citation.cfm?id=1251375.1251388

[8] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, B. B. Kang, Vigilare: toward snoop-based kernel integrity monitor, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS ’12, ACM, 2012, pp. 28–37. doi:10.1145/2382196.2382202. URL http://dl.acm.org/citation.cfm?id=2382202

[9] H. Lee, K. Advanced, T. Kaist, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, B. B. Kang, KI-Mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object, in: Proceedings of the 22nd USENIX Conference on Security, 2013, pp. 511–526.

[10] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, W. Shi, CPU transparent protection of OS kernel and hypervisor integrity with programmable DRAM, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, ACM, 2013. doi:10.1145/2508148.2485956.

[11] Vmware security advisories, http://www.vmware.com/security/advisories, last accessed Feb 4, 2015.

[12] Advisories, publicly released or pre-released, http://xenbits.xen.org/xsa/, last accessed Feb 4, 2015.

[13] A. T. Rafal Wojtczuk, Joanna Rutkowska, Xen 0wning trilogy, http://invisiblethingslab.com/itl/Resources.html, last accessed Feb 4, 2015 (2008).

[14] N. Elhage, Virtunoid: A kvm guest host privilege escalation exploit, in: BlackHat USA, 2011.

[15] K. Kortchinsky, Cloudburst: Hacking 3d (and breaking out of vmware), in: BlackHat USA, 2009.

[16] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, B. B. Kang, ATRA: Address Translation Redirection Attack Against Hardware-based External Monitors, in: Proceedings of the 2014 ACM Conference on Computer and Communications Security, CCS '14, ACM, 2014, pp. 167–178. doi:10.1145/2660267.2660303. URL http://dl.acm.org/citation.cfm?id=2660267.2660303

[17] D. Bovet, M. Cesati, Understanding The Linux Kernel, Oreilly & Associates Inc, 2005.

[18] W. Mauerer, Professional Linux Kernel Architecture, Wrox Press Ltd., Birmingham, UK, UK, 2008.

[19] N. L. Petroni, Jr., M. Hicks, Automated detection of persistent kernel control-flow attacks, in: Proceedings of the 14th ACM conference on Computer and communications security, CCS '07, ACM, New York, NY, USA, 2007, pp. 103–115. doi:10.1145/1315245.1315260. URL http://doi.acm.org/10.1145/1315245.1315260

[20] A. Baliga, V. Ganapathy, L. Iftode, Automatic inference and enforcement of kernel data structure invariants, in: Proceedings of the 2008 Annual Computer Security Applications Conference, ACSAC '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 77–86. doi:10.1109/ACSAC.2008.29. URL http://dx.doi.org/10.1109/ACSAC.2008.29

[21] B. Dolan-Gavitt, A. Srivastava, P. Traynor, J. Giffin, Robust signatures for kernel data structures, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS ’09, ACM, New York, NY, USA, 2009, pp. 566–577. doi:10.1145/1653662.1653730. URL http://doi.acm.org/10.1145/1653662.1653730
[22] Z. Lin, J. Rhee, X. Zhang, D. Xu, Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures, in: Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS’11), San Diego, CA, 2011.
[23] B. D. Payne, M. Carbone, M. Sharif, W. Lee, Lares: An architecture for secure active monitoring using virtualization, in: Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP ’08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 233–247. doi:10.1109/SP.2008.24. URL http://dx.doi.org/10.1109/SP.2008.24
[24] X. Xiong, D. Tian, P. Liu, P.: Practical protection of kernel integrity for commodity os from untrusted extensions, in: In: Proceedings of the Network and Distributed System Security Symposium, NDSS 11. The Internet Society, 2011.
[25] S. Liakh, M. Grace, X. Jiang, Analyzing and improving linux kernel memory protection: A model checking approach, in: Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC ’10, ACM, New York, NY, USA, 2010, pp. 271–280. doi:10.1145/1920261.1920301. URL http://doi.acm.org/10.1145/1920261.1920301
[26] L. Torvalds, “re: [patch 0/2] xfs: Fix some new memory allocation failures., http://yarchive.net/comp/linux/vmalloc.html, last accessed Feb 4, 2015 (2003).
[27] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, 3rd Edition, O’Reilly Media, Inc., 2005.
[28] J. Rhee, D. Xu, Livedm: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging, Tech. rep., West Lafayette, IN (February 2010).
[29] D. Hellström, SnapGear Linux for LEON, Gaisler Research (November 2008).
[30] M. I. Sharif, W. Lee, W. Cui, A. Lanzi, Secure in-vm monitoring using hardware virtualization, in: Proceedings of the 16th ACM conference on Computer and communications security, CCS ’09, ACM, New York, NY, USA, 2009, pp. 477–487. doi:10.1145/1653662.1653720. URL http://doi.acm.org/10.1145/1653662.1653720
[31] Z. Wang, X. Jiang, W. Cui, P. Ning, Countering kernel rootkits with lightweight hook protection, in: Proceedings of the 16th ACM conference on Computer and communications

security, CCS ’09, ACM, New York, NY, USA, 2009, pp. 545–554. doi:10.1145/1653662.1653728. URL http://doi.acm.org/10.1145/1653662.1653728

[32] L. Project, Virtual machine introspection: Fast, portable, simple, http://http://libvmi.com, last accessed Nov 20, 2018 (2005).

[33] X. Wang, Y. Chen, Z. Wang, Y. Qi, Y. Zhou, Secpod: A framework for virtualization-based security systems, in: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’15, USENIX Association, Berkeley, CA, USA, 2015, pp. 347–360. URL http://dl.acm.org/citation.cfm?id=2813767.2813793

[34] A. Prakash, H. Yin, Z. Liang, Enforcing system-wide control flow integrity for exploit detection and diagnosis, in: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS ’13, ACM, New York, NY, USA, 2013, pp. 311–322. doi:10.1145/2484313.2484352. URL http://doi.acm.org/10.1145/2484313.2484352

Figure 1: Linux kernel paging structures

Figure 2: Characteristics of Linux kernel memory layout: *Physical Mappings* which constitute 88.7% of kernel memory are both linear and persistent. *Fixmaps* region is persistent but not necessarily linear. *Persistent Mappings* and *vmalloc* region are neither linear nor persistent

Figure 3: Amended HW architecture (SPARC) Overview: The registers involved with address space management (Context Register, Context Table Pointer Register) are exported to external monitor

Figure 4: Context Switch Verification Scheme: For each inter-process context switching event that accompanies an address space change, external monitor is given a PTBR Change Event. In turn, the PGD_START/END Registers are adjusted to accommodate the PTBR change. This adjusting procedure is represented by `adjust()`. In addition, a hash check function `check()` is performed on the `PGD` that is pointed by the newly changed value in PTBR to ensure the integrity of the now-active PGD

Figure 5: Kernel Memory Layout of Prototype Host (SPARC Leon3 Processor with Snapgear Linux)

Figure 6: Time Constraints for Memory Mapping Integrity Verification

Table 1: Execution time of our verification scheme

Entry Size	# `PGD` Entry	Execution Time
32bit	5	4.34 μs

32bit	512	102.26 μs
32bit	1024	202.26 μs
64bit	1024	400.66 μs